Xpert.press

Volker Gruhn
Daniel Pieper
Carsten Röttgers

MDA®

Springer

Xpert.press

Die Reihe **Xpert.press** vermittelt Professionals in den Bereichen Softwareentwicklung, Internettechnologie und IT-Management aktuell und kompetent relevantes Fachwissen über Technologien und Produkte zur Entwicklung und Anwendung moderner Informationstechnologien.

Volker Gruhn · Daniel Pieper
Carsten Röttgers

MDA®

Effektives Software-Engineering mit UML 2® und Eclipse ™

Mit 293 Abbildungen

Springer

Volker Gruhn
Lehrstuhl für
Angewandte Telematik/e-Business
Universität Leipzig
Klostergasse 3
04109 Leipzig
gruhn@ebus.informatik.uni-leipzig.de

Daniel Pieper
Carsten Röttgers
adesso AG
Stockholmer Allee 24
44269 Dortmund
dp@mda-buch.info
cr@mda-buch.info

Bibliografische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.

ISSN 1439-5428
ISBN-10 3-540-28744-2 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-28744-5 Springer Berlin Heidelberg New York

Dieses Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere die der Übersetzung, des Nachdrucks, des Vortrags, der Entnahme von Abbildungen und Tabellen, der Funksendung, der Mikroverfilmung oder der Vervielfältigung auf anderen Wegen und der Speicherung in Datenverarbeitungsanlagen, bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Eine Vervielfältigung dieses Werkes oder von Teilen dieses Werkes ist auch im Einzelfall nur in den Grenzen der gesetzlichen Bestimmungen des Urheberrechtsgesetzes der Bundesrepublik Deutschland vom 9. September 1965 in der jeweils geltenden Fassung zulässig. Sie ist grundsätzlich vergütungspflichtig. Zuwiderhandlungen unterliegen den Strafbestimmungen des Urheberrechtsgesetzes.

Springer ist ein Unternehmen von Springer Science+Business Media
springer.de

© Springer-Verlag Berlin Heidelberg 2006
Printed in Germany

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften. Text und Abbildungen wurden mit größter Sorgfalt erarbeitet. Verlag und Autor können jedoch für eventuell verbliebene fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.

Satz: Druckfertige Daten der Autoren
Herstellung: LE-TEX, Jelonek, Schmidt & Vöckler GbR, Leipzig
Umschlaggestaltung: KünkelLopka Werbeagentur, Heidelberg
Gedruckt auf säurefreiem Papier 33/3142 YL – 5 4 3 2 1 0

Vorwort

„Simple things should be simple and complex things should be possible."

Alan Kay

Informationstechnik ist über den Lauf der Jahre zum Rückgrat der Geschäftswelt geworden: kaum eine kommerzielle Transaktion läuft heute nicht IT-gestützt ab, nahezu jeder Euro, der seinen Besitzer wechselt, wird elektronisch erfasst. Die Geschäftsprozesse vieler Unternehmungen entwachsen den vier Wänden ihrer Bürogebäude oder Produktionsstätten und spielen sich in Zeiten von Globalisierung und sich öffnenden Märkten zunehmend auch auf internationalem Parkett ab. Solch internationales Treiben ist dabei nicht länger auf große Konzerne beschränkt, sondern wird immer mehr auch für kleine und mittelständische Unternehmen zum Tagesgeschäft. Die daraus resultierenden Anforderungen erfordern eine entsprechende IT mit maßgeschneiderten Lösungen, die die individuellen Bedürfnisse der Unternehmen berücksichtigen.

Das Problem

Wir können nicht wie wir wollen. Man mag es ja kaum noch hören, aber die Software-Industrie steckt immer noch in der bereits vor Jahrzehnten ausgerufenen Software-Krise. Gewiss, in puncto Effizienz, Effektivität und Produktivität sind Fortschritte erzielt worden, aber wenn wir ehrlich sind, sind es eher Schrittchen, von denen wir da sprechen und weniger die erhofften Sprünge. Die Bestandsaufnahme der Software-Projektlandschaft fällt dementsprechend fatal aus: Falls ein Projekt tatsächlich überlebt, dann selten *in Time* und noch viel seltener *in Budget*. Bei der Qualität wird es dann schließlich ganz dünn, sie fällt meist gleich als Erstes über Bord, sobald Time und Budget in Gefahr scheinen.

Dabei wird Software immer komplexer, die Ansprüche an ihre Leistungsfähigkeit, Zuverlässigkeit, Sicherheit und Skalierbarkeit bei stetig steigendem Kosten- und Zeitdruck gleichzeitig immer größer. Ein *Circulus Diaboli*, bei dem man am liebsten den Kopf in den Sand stecken möchte.

Dumm nur, wenn der Kunde dieses Spiel nicht mehr mitspielen mag und selbst aktiv nach Auswegen sucht. In letzter Zeit werden dabei immer häufiger Outsourcing und Near-/Offshoring-Maßnahmen diskutiert. Auch wenn Yourdon in [You05] attestiert, dass sich die Entwicklung von betriebswirtschaftlicher Individualsoftware weniger zu solchen Maßnahmen eignet als etwa die von wissenschaftlich oder technisch geprägter Software, fällt sein Resümee am Ende ernüchternd aus: Immer öfter gewinnen spezialisierte Offshorer auch in diesem Bereich gegen Dienstleister aus der *alten Welt* und zwar nicht nur bei den Kosten, sondern auch bei Termintreue und Qualität. Geben wir es ruhig zu: die westliche IT-Industrie ist auch hier überrumpelt worden. Über die Hälfte aller CMM5-zertifizierten Unternehmen kommt mittlerweile aus Indien.

Für die Software entwickelnden Unternehmen heißt es also endlich aktiv zu werden und nach Lösungen für eine effizientere Entwicklung und Pflege von Software zu suchen. Aussprüche wie „Das Ende der Software-Entwicklung wie wir sie kennen“ scheinen so gerade rechtzeitig den nächsten Anlauf auf die Brook´sche Silverbullet einzuläuten. Sollten wir nicht eigentlich gelernt haben, dass es selbige nicht geben kann? Nun, glauben wir zumindest Frankel [Fra04] und Greenfield et al. [GSCK04], so stehen wir allen Unkenrufen zum trotz kurz vor dem Eintritt in eine Phase zunehmender Professionalisierung der Softwareentwicklung.

Frankel wählt dazu den Begriff der *Software-Industrialisierung*, Greenfield spricht in diesem Zusammenhang gar von *Software-Fabriken*. Uneins ist man sich zurzeit noch, wie die konkrete Manifestierung dieser Ideen auszusehen hat. Einigkeit scheint hingegen darüber zu herrschen, dass aus technischer Sicht die Softwareentwicklung zukünftig architekturzentriert, aspektorientiert, domänenspezifisch und modellgetrieben erfolgen muss.

Der Vorschlag der Object Management Group

Eine ganzheitliche Vorgehensweise, die versucht durch die synergetische Verwendung bekannter Prinzipien und Methoden den neuen Anforderungen zu begegnen, ist die Model-Driven Architecture (MDA) der OMG, die bereits seit dem Jahr 2000 einen der großen Hoffnungsträger für den Weg aus der Krise darstellt.

Sie beschreibt den Rahmen für ein modellgetriebenes Vorgehen, bei dem Modelle in den Mittelpunkt des Entwicklungsprozesses

gerückt werden. Kern dieser Architektur ist ein Konzept, das zwischen plattform-unabhängigen und plattform-spezifischen Modellen unterscheidet und es so ermöglicht, die Spezifikation der Funktionalität eines Systems von der Spezifikation, wie diese Funktionalität auf eine spezifische Plattform implementiert wird, zu trennen. Der Übergang von der fachlich getriebenen Spezifikation zur ausführbaren Anwendung erfolgt dabei möglichst automatisiert durch geeignete Transformationswerkzeuge.

Versprochen wird sowohl Kostensenkung durch die generative Erzeugung von weiten Teilen der Anwendung und beschreibender Modelle sowie die vereinfachte Wiederverwendung horizontaler Querschnittskomponenten in der Entwicklung von Folgeanwendungen. Durch die Erstellung technologieunabhängiger Modelle soll zudem der Forderung der Unternehmen nach Konservierung der Fachlichkeit entsprochen werden, indem die fachlichen Modelle ihre Gültigkeit, auch bei sich ändernden unterliegenden Technologielandschaften, behalten und so die aktuelle Dominierung der Fachlichkeit durch die kurzen technologischen Änderungszyklen durchbrochen wird.

Die notwendigen Mittel zur Umsetzung

Um die eben dargestellten Prinzipien anzuwenden, bedarf es einer maßgeschneiderten unterstützenden Infrastruktur, die unter anderem die benötigten domänenspezifischen Sprachen, die geeigneten Prozesse sowie die technischen Werkzeuge zur Unterstützung des Automatisierungsansatzes liefert bzw. ermöglicht. Bislang waren diese Werkzeuge proprietär und herstellergebunden bzw. mit hohen Eigenentwicklungskosten verbunden, die gerade für kleine und mittelständische Unternehmen (KMUs) mit zu hohem Risiko und Investitionsaufwand zu Buche schlagen, um ernsthaft in Erwägung gezogen zu werden. Mittlerweile existiert jedoch eine Vielzahl offener Standards und Open-Source Frameworks, die einen Versuch der Umsetzung in Eigenregie in neuem Glanz erscheinen lassen.

Mit den Modellierungsstandards der OMG, namentlich des UML und MOF 2.x Stacks, liegt nun auch der benötigte konzeptuelle Überbau vor; domänenspezifische Erweiterungen sind in der Konzeption bzw. kurz vor der Fertigstellung. Open-Source Projekte wie Eclipse, die von vielen großen Firmen unterstützt werden, bilden ganze Ökosysteme frei verfügbarer technischer Bausteine, diese Konzepte auch auf praktischer Ebene umsetzen zu können.

Das vorliegende Buch …

… nimmt zum einen die theoretischen Konzepte unter die Lupe und versetzt den Leser so in die Lage die Eignung der MDA für die

individuelle Situation seines Unternehmens besser einordnen und bewerten zu können, liefert zum anderen jedoch auch konkrete Betrachtungen zur praktischen Umsetzung mittels frei verfügbarer Techniken und Technologien. Besonderer Fokus wird dabei auf die Verwendung der Unified Modeling Language (UML) als Modellierungssprache der Wahl sowie auf das Eclipse-Projekt als technische Basis zur Umsetzung gelegt. Im Verlaufe der Untersuchung der Randaspekte der MDA werden weiterhin viele weitere Konzepte und Technologien des modernen Software-Engineerings aufgegriffen und vertieft, sodass die Lektüre auch dann lohnt, wenn die Einführung MDA im eigenen Hause nicht kurz vor der Tür steht.

Aufgrund der Vielfalt der Themen können viele hochinteressante Aspekte der MDA jedoch trotzdem nur am Rande behandelt werden. Wir hoffen jedoch, dass bei der getroffenen Auswahl auch die für Ihre individuelle Situation passenden Themenkomplexe enthalten sind und wünschen Ihnen in diesem Sinne *Viel Spaß beim Lesen.*

Literatur

[Fra04] David S. Frankel: *Software Industrialization and the New IT*. In: David S. Frankel und John Parodi (Hrsg.): The Mda Journal – Model Driven Architecture Straight From the Masters. Meghan-Kiffer, 2004.

[GSCK04] Jack Greenfield, Keith Short, Steve Cook und Stuart Kent: *Software Factories – Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, 2004.

[You04] Edward Yourdon: *Outsource – Competing in the Global Productivity Race*. Prentice Hall, 2004.

Inhaltsverzeichnis

1 EINLEITUNG 1

1.1 An wen wendet sich dieses Buch 1
1.1.1 Entscheider/Manager/Projektleiter 1
1.1.2 Berater 2
1.1.3 Architekten und Entwickler 2

1.2 Ziele des Buches 2
1.2.1 Wie lässt sich die MDA einordnen? 3
1.2.2 Darstellung der Konzepte der MDA 3
1.2.3 Koordination und Kombination 3
1.2.4 genua – prototypisches MDA Framework 4
1.2.5 Fazit – Ist MDA endlich die silberne Kugel? 4

1.3 Überblick und Leitfaden zum Lesen 4

1.4 Konventionen 6

1.5 Weitere Informationen 7

2 MDA – ÜBERBLICK UND ORIENTIERUNG 9

2.1 Motivation modellgetriebener Ansätze 9
2.1.1 Die Geschichte der Softwareentwicklung – ein historischer Abriss 11
2.1.2 Die Gegenwart 14
2.1.3 Akute Probleme bei der Software-Erstellung 16
2.1.4 Die Idee modellgetriebener Ansätze 19

2.2 Die Model-Driven Architecture (MDA) 21
2.2.1 Ziele der MDA 21
2.2.2 Die Vorgaben der Object Management Group (OMG) 23
2.2.3 Metamodell der zentralen MDA-Begrifflichkeiten 25
2.2.4 Standards im Dunstkreis der MDA 31

2.3 Ideen, Anleihen und verwandte Ansätze 32

2.3.1 Plattformunabhängigkeit ... 34
2.3.2 Ausführbare Modelle ... 35
2.3.3 Klassen, Komponenten und Frameworks ... 36
2.3.4 Musterorientierung ... 37
2.3.5 Architekturzentrierung ... 39
2.3.6 Aspektorientierung ... 40
2.3.7 Konvergenz ... 41
2.3.8 Domain Engineering ... 43
2.3.9 Generative Programming ... 45
2.3.10 Software-Factories et al. ... 46

2.4 Pragmatische Sichten auf MDA ... 47
2.4.1 MDA-light ... 49
2.4.2 Warum jetzt? ... 49

2.5 Also … ... 51

3 MODELLIERUNG ... 57

3.1 Grundlagen der Modellierung ... 57
3.1.1 Sketch-Modelle ... 63
3.1.2 Formale Modelle ... 65
3.1.3 Kurze Rekapitulation ... 71

3.2 Unified Modeling Language (UML) ... 73
3.2.1 Historisches ... 73
3.2.2 UML – Die Sprache der Model-Driven Architecture ... 75
3.2.3 UML-Spezifikationen ... 80

3.3 Metamodellierung ... 84
3.3.1 Meta? – Grundlagen ... 84
3.3.2 Meta Object Facility 2 (MOF 2) ... 87
3.3.3 Beispiel für ein Metamodell ... 92
3.3.4 UML-Profile ... 94

3.4 UML-Repository ... 98

3.5 UML-Action-Semantics ... 101

3.6 OCL – Object Constraint Language ... 106
3.6.1 Grundlagen – Was ist OCL? ... 106
3.6.2 Zuordnung von OCL-Ausdrücken zu Modellelementen ... 108
3.6.3 Anwendungsmöglichkeiten von OCL ... 110

4 MODELLE DER MDA ... 119

4.1 Lebenszyklus von MDA-Modellen ... 120

4.2 Computation Independent Model (CIM) ... 122

4.3 Plattform Independent Model (PIM) 126
4.4 Architecture Metamodel (AMM) 130
4.5 Platform Description Model (PDM) 139
4.6 Platform Specific Model (PSM) 141

5 TRANSFORMATION 149

5.1 Einführung 149
5.2 Anwendungsfälle für Transformationen 151
5.3 Modell-zu-Modell Transformationen 153
5.3.1 Das Schema metamodellbasierter Modell-transformationen 154
5.3.2 Beispiel: UML 2.0 PIM zu Java PSM 157
5.3.3 Implementierungs-Strategien für Transformationen 164
5.4 Modell-zu-Text Transformationen 167
5.4.1 Fortführung des Beispiels: Java PSM zu Java Code 168
5.4.2 … vom Modell zum Text 171
5.4.3 Synchronisation von Modellen und Code 173
5.5 PIM → Code vs. PIM → PSM → Code 178

6 KOORDINATION UND KOMBINATION 183

6.1 Grundlagen und Vogelperspektive 184
6.1.1 Das Prozessmodell von oben 185
6.1.2 Domain Engineering 188
6.1.3 Application Engineering 191
6.2 Aktivitäten und Artefakte 191
6.2.1 Domäne qualifizieren 191
6.2.2 Domäne analysieren 194
6.2.3 Framework implementieren 196
6.2.4 System-Modellierung 197
6.2.5 Transformation 198
6.2.6 Feedback 199
6.3 Rollen und (neue) Aufgaben 201
6.3.1 Domain Engineering 202
6.3.2 Application Engineering 206
6.3.3 Fazit 211
6.4 Einführung von MDA ins Unternehmen 211
6.4.1 Ad-hoc-Vorgehen oder Iterative Einführung? 213
6.4.2 Pilotprojekte 216

6.4.3 Fazit ... 217

6.5 Anpassen bestehender Organisationsstrukturen ... 218
6.5.1 Drei mögliche Organisationsmodelle ... 220

6.6 Best Practices und Gefährliches ... 223
6.6.1 Iterativ-Inkrementelle Softwareentwicklung ... 223
6.6.2 Best Practices … ... 230
6.6.3 … und Gefährliches ... 237

7 VORSTELLUNG DES FALLBEISPIELS ... 247

7.1 Ausgangssituation ... 248

7.2 Modell des Geschäftssystems ... 250
7.2.1 Modellierungsfokus ... 251
7.2.2 Organisationseinheiten ... 252
7.2.3 Geschäftspartner ... 253
7.2.4 Geschäftsanwendungsfälle der aktiven Geschäftspartner ... 254
7.2.5 Weitere unterstützende Geschäftsanwendungsfälle ... 256
7.2.6 Geschäftsmitarbeiter/Akteurmodelle ... 256
7.2.7 Geschäftsprozesse ... 257
7.2.8 Essenzbeschreibungen der Geschäftsanwendungsfälle ... 259
7.2.9 Ablaufmodelle der Geschäftsanwendungsfälle ... 261
7.2.10 Ablauf Geschäftsprozess ... 262
7.2.11 Geschäftsklassenmodell ... 263

7.3 Ergebnis der Geschäftsprozessmodellierung ... 264

7.4 Das weitere Vorgehen ... 265

8 PROJEKTPLANUNG ... 269

8.1 Exploratory 360º ... 269
8.1.1 Systemanforderungen ... 270
8.1.2 Ein erster Releaseplan der Anwendung ... 272
8.1.3 Anforderungen an „genua“ ... 273
8.1.4 Releaseplan des genua Frameworks ... 277
8.1.5 Erste Projektpläne ... 277

8.2 Technologie-Plan ... 279
8.2.1 Das Eclipse-Projekt ... 279
8.2.2 Hibernate ... 283
8.2.3 Graphical Editing Framework (GEF) ... 289
8.2.4 Eclipse Modeling Framework (EMF) + Eclipse UML2 ... 291
8.2.5 JavaServer Faces ... 300
8.2.6 Apache Beehive ... 304

8.2.7 jBPM 311

9 PROJEKTDURCHFÜHRUNG 327

9.1 Architektur von genua Anwendungen 327

9.2 Dialoge und Kontrollflüsse 330
9.2.1 genua Platform Independent Pageflow Profile (gPIPfP) 331
9.2.2 Das erste Modell des Projekts „M&M online“ 339
9.2.3 genua Beehive Pageflow Profile (gBPfP) 345
9.2.4 genua JSF Metamodel (gJSFMM) 353
9.2.5 genua Platform Independent Workflow Profile (gPIWfP) 356
9.2.6 Ein weiteres Modell des M&M-Projektes 358
9.2.7 genua jBPM Profil 360
9.2.8 Transformation von gPIWfP nach gjBPM 363
9.2.9 Vom Modell zum lauffähigen Workflow 364

9.3 Geschäftslogik/Services 371
9.3.1 Beehive Controls 371
9.3.2 Realisierung der Services als SLSBs 376

9.4 Persistenz 378
9.4.1 genua Platform Independent Persistence Profile (gPIP) 379
9.4.2 genua Hibernate Persistence Profile (gHPP) 382
9.4.3 Strategie zur Abbildung Hibernate → gHPP 386
9.4.4 Anbindung an die Serviceschicht 388

9.5 genua Model2Model-Transformator (gM2M) 389
9.5.1 Essenzieller Ablauf 389
9.5.2 genua ATLAS Transformation Language (gATL) 390
9.5.3 Verarbeitung von gATL Transformationsmodulen 394
9.5.4 Beispiel 396
9.5.5 Konstruktion des Zwischenmodells 399
9.5.6 Deserialisierung des Quellmodells 403
9.5.7 Durchführung der Transformation mittels Jython 404
9.5.8 Durchführung einer Transformation mittels Ant 408

9.6 genua Model2Text-Transformator (gM2T) 411
9.6.1 Java Emitter Templates (JET) 413
9.6.2 Modell-Fassaden 414
9.6.3 Zusammenspiel der Komponenten 415
9.6.4 Literaturempfehlungen 417

10 /LOST+FOUND 421

10.1 Bringt MDA einen ROI? – Die etwas andere Sichtweise. .421

10.2 Software-Factories vs. MDA ... 423
10.2.1 Was sind Software-Factories? ... 424
10.2.2 Reizwort „UML“ ... 425
10.2.3 MDA – was fehlt? ... 425
10.2.4 Andere Meinungen zum Thema ... 426

11 ENDE GUT – ALLES GUT? ... 433

11.1 Was fehlt bzw. ist zu tun? ... 434

11.2 Was wird? ... 436
11.2.1 Wie lange dauert es noch bis MDA zur Commodity wird? ... 436
11.2.2 Woran könnte die MDA noch scheitern? ... 438

11.3 Proof-of-Concept erfolgreich? ... 442
11.3.1 Der MDA-Prozess – in Sicht? ... 442
11.3.2 genua – Prototypisches MDA-Framework ... 443

11.4 … schließende Worte ... 444

A UML-SCHNELLREFERENZ ... 447

A.1 Strukturdiagramme ... 447
A.1.1 Klassendiagramm ... 447
A.1.2 Objektdiagramm ... 454
A.1.3 Paketdiagramm ... 455
A.1.4 Komponentendiagramm ... 457
A.1.5 Verteilungsdiagramm ... 460
A.1.6 Kompositionsstrukturdiagramm ... 461

A.2 Verhaltensdiagramme ... 464
A.2.1 Use-Case Diagramm ... 465
A.2.2 Aktivitätsdiagramm ... 467
A.2.3 Zustandsautomat ... 474
A.2.4 Sequenzdiagramm ... 479
A.2.5 Interaktions-Übersichts Diagramm ... 483
A.2.6 Kommunikationsdiagramm ... 484
A.2.7 Zeitdiagramm ... 486

A.3 Literaturtipps ... 487

B OOGPM ... 491

B.1 Einleitung und Übersicht ... 491

B.2 Organisationseinheiten modellieren ... 494

B.3 Aktive Geschäftspartner identifizieren ... 495

B.4 Geschäftsanwendungsfälle der aktiven Geschäftspartner identifizieren 496
B.5 Geschäftsmitarbeiter identifizieren und Akteurmodell entwickeln 498
B.6 Geschäftsprozesse definieren 499
B.6.1 GAF-Abläufe modellieren 499
B.7 Literaturempfehlungen 500

1 Einleitung

„If all you have is a hammer, everything looks like a nail."

Bernard Baruch

1.1 An wen wendet sich dieses Buch

Das vorliegende Buch ist so aufgebaut, dass es Aspekte für diejenigen Leser enthält, die sich „nur" über die Wurzeln und theoretischen Grundlagen der Model-Driven Architecture informieren wollen, aber auch den Bogen zu deren praktischer Anwendung schlägt, sodass auch Praktiker, die Tipps zur konkreten Umsetzung erwarten, sich in diesem Buch wiederfinden werden.

Speziell werden folgende Themenkomplexe für die folgenden Interessengruppen behandelt:

1.1.1 Entscheider/Manager/Projektleiter

Für die Unentschiedenen, die nicht sicher sind, ob die Vorgehensweisen der MDA zur Bewältigung ihrer individuellen Anforderungen prinzipiell geeignet sind, bietet das vorliegende Buch erste Einblicke und die notwendige Grundlage zur fundierten Entscheidungsfindung. Für alle, die diese Entscheidung bereits getroffen haben oder kurz davor stehen, wird konkret auf die Auswirkungen eingegangen, die eine Einführung auf die bestehende Unternehmensorganisation hat bzw. haben sollte und bietet Tipps zur Adaption bestehender Entwicklungsprozesse.

Auch als „Know-how Update", um über die Themen auf dem Laufenden zu bleiben, die aktuell die Softwareentwicklung beschäftigen, ist das vorliegende Buch bestens geeignet. Die Basiskonzepte

und Ideen der MDA werden ausführlich erläutert und in den Kontext des aktuellen State-of-the-Art der Softwareentwicklung eingeordnet.

1.1.2 Berater

Für die Berater unter Ihnen liefert das Buch das notwendige Hintergrundwissen zu den, oft nur als Buzzwords missbrauchten, Konzepten eines der aktuell interessantesten Bereiche des Software-Engineerings, das zur fundierten Einschätzung und Einordnung dieser Ideen in der Praxis benötigt wird, um eine professionelle Beratungsleistung in diesem Themenkomplex bieten zu können. Anhand von Beispielen wird gezeigt, wie den Anforderungen des IT-Alltags mit den Werkzeugen der MDA begegnet werden kann. Die vorgestellten Bausteine können auch in Ansätzen, die nicht den Stempel MDA tragen, verwendet werden und bilden so eine sinnvolle Ergänzung jedes Consulting-Toolkits.

Last-but-not-least werden Antworten auf häufig – und berechtigt – gestellte Fragen zu Machbarkeit und ROI des Ansatzes gegeben, die als Argumentationshilfe in Kundengesprächen wertvolle Dienste leisten können.

1.1.3 Architekten und Entwickler

Anhand eines Fallbeispiels werden die im ersten Teil des Buches theoretisch erarbeiteten Konzepte und Forderungen nach weitestgehender Werkzeugunterstützung des MDA-Prozesses durch die Erarbeitung eines Beispielframeworks umgesetzt, das ganz ohne proprietär kommerzielle Bausteine auskommt, und ausschließlich mittels Open-Source Technologien realisiert werden kann. Der Fokus wird dabei vor allem auf das Eclipse-Projekt [Eclipse] sowie dessen untergeordnete Teilprojekte gelegt, da diese eine fast vollständige Abdeckung der notwendigen Bausteine eines solchen Frameworks liefern.

Ein mit der MDA-Infrastruktur eng verbundener Überblick über die aktuellen Softwaretools und Bibliotheken, die zur Realisierung der vertikalen und horizontalen Domänenaspekte herangezogen werden, rundet den Themenkomplex für die Praktiker unter den Lesern ab.

1.2 Ziele des Buches

Im Sinne eines *Proof-of-Concept* sollen in diesem Buch zuerst die Ziele untersucht werden, zu deren Lösung die MDA antritt und anschließend die Betrachtung nicht unterschlagen werden, ob der Ansatz diese Versprechungen mit den aktuell verfügbaren Techno-

logien prinzipiell erfüllen kann. Dabei sollen vor allem die folgenden Themenbereiche näher betrachtet werden:

1.2.1 Wie lässt sich die MDA einordnen?

Im Ansatz MDA finden sich viele Anleihen bekannter und weniger bekannter Prinzipien, Technologien und Vorgehensweisen aus dem weiten Feld des Software-Engineering. Wir wollen versuchen, diese Anleihen zu identifizieren und ihre Bedeutung innerhalb der Model-Driven Architecture näher zu beleuchten. Da einige Bestandteile bereits in der Vergangenheit große Versprechungen bezüglich der Effektivitätssteigerung der Softwareentwicklung nicht oder nur teilweise halten konnten, wollen wir außerdem versuchen zu klären, ob eine Kombination dieser Einzelteile zum jetzigen Zeitpunkt eventuell größere Aussichten auf Erfolg haben kann und die Gründe für unsere Schlussfolgerungen erläutern.

1.2.2 Darstellung der Konzepte der MDA

Hier sollen die technischen Grundlagen der MDA vorgestellt und die hinterliegenden Prinzipien erläutert werden. Modelle und (Meta-)Modellierung, spezifische Modellarten der MDA, der Plattformbegriff, Transformationen und Transformationsbeschreibungen sowie UML sind nur einige Begriffe, die verstanden und angewendet werden müssen, um die Vorstellungen der OMG umsetzen zu können. Wir wollen diese Begriffe in ihre übergeordneten Themenkomplexe einordnen und sowohl die spezifischen Einzelheiten, aber auch die großen Zusammenhänge ausführlich erläutern, da sie das Fundament des zweiten Teils der Betrachtungen bilden: der Untersuchung der Tragfähigkeit der MDA durch Konzeption und ausschnittsweise Implementierung eines Frameworks namens *genua* (vgl. Abschnitt 1.2.4) anhand eines Projekt-Fallbeispiels.

1.2.3 Koordination und Kombination

Nicht nur die Machbarkeit der technologischen Umsetzung der MDA-Prinzipien ist zu zeigen, auch die organisatorischen Aspekte müssen näher untersucht werden. So setzt die MDA a priori weder einen dezidierten Software-Entwicklungsprozess als Grundlage voraus, noch werden von der OMG selbst irgendwelche Vorschläge bezüglich des zu verwendenden Prozessmodells gemacht. Zu untersuchen ist also, was ein Prozess mindestens leisten muss, um die zusätzlichen Anforderungen, die die Vorgehensweise erfordert, erfüllen zu können. Beantwortet werden sollen also unter anderem folgende Fragen: Welche Anforderung stellt MDA an Organisation

und Prozesse? Welche Implikationen ergeben sich? Und wie können bestehende Strukturen und Vorgehensmuster so angepasst werden, dass diese Anforderungen erfüllt werden.

1.2.4 genua – prototypisches MDA Framework

Hier wird versucht die theoretisch erarbeiteten Anforderungen an eine technische Infrastruktur zur Stützung der MDA prototypisch umzusetzen. In einem Fallbeispiel wird unter dem Namen *genua* ein MDA-Framework entwickelt, das aus den geforderten Tools wie Generatoren, Transformatoren, Profilen usw. besteht, die in den vorangegangenen Kapiteln als unverzichtbar identifiziert wurden.

Die Technologien, die zur Umsetzung ausgewählt wurden, werden erklärt und das Zusammenspiel im MDA-Prozess anhand eines zusammenhängenden Fallbeispiels auszugsweise erläutert. Untersucht wird so die Tragfähigkeit und Integrierbarkeit der zur Zeit verfügbaren Technologielandschaft gegenüber Ansprüchen der MDA. Hierbei werden dezidiert nur frei verfügbare Open-Source Elemente und frei verfügbare Standards betrachtet.

1.2.5 Fazit – Ist MDA endlich die silberne Kugel?

Die in den vorigen Kapiteln gewonnenen Einsichten und Erkenntnisse werden hier in einer komprimierten Form zusammengefasst und die Tragfähigkeit der „Idee MDA“ von den Autoren anhand der gemachten Erfahrungen und theoretischen Überlegungen bewertet. Zusätzlich sollen ausgewählte Schwachstellen noch einmal explizit genannt und ein Blick in die Glaskugel auf die nähere Zukunft der MDA versucht werden.

1.3 Überblick und Leitfaden zum Lesen

Die Konzeption des Buchaufbaus folgt einer Dreiteilung: Teil I (Kapitel 2–5) beschreibt die theoretischen Grundlagen des Ansatzes (die *Architektur* der MDA). Im zweiten Teil (Kapitel 6) werden die Auswirkungen der vorgestellten Konzepte auf die Prozesse und die Organisation von Unternehmen näher beleuchtet. Teil III (Kapitel 7–9) bringt die Ideen zur Anwendung und setzt sie mit konkreten Technologien zur Umsetzung in Beziehung.

Es folgt ein Überblick über den Inhalt der einzelnen Kapitel:

Kapitel 2: MDA – Überblick und Orientierung

Hier werden die grundlegenden Konzepte und Begriffe erläutert, auf denen die MDA basiert. Außerdem wird der Ansatz in den „historischen Kontext“ des Software-Engineering eingebettet und die Beziehungen zwischen „alten“ Ideen und „neuer“ MDA hergestellt.

Kapitel 3: Modellierung

Als einem der Hauptwerkzeuge der MDA wird der Modellierung ein eigenes Kapitel gewidmet. Hier werden die entsprechenden grundlegenden Begriffe und Konzepte eingeführt und das Werkzeug der OMG, die Unified Modeling Language (UML) sowie deren Spezifikationen unter die Lupe genommen. Weitere behandelte Punkte sind: formale Modellierung, Metamodellierung, MOF, Action Semantics, UML Repository usw.

Kapitel 4: Modelle der MDA

Hier wird der „Lebenszyklus“ der Modellarten der MDA betrachtet und ein erstes durchgängiges Beispiel für konkrete Modelle vom anwendungs-unahängigen Modell (CIM) bis zur fertigen Applikation gezeigt.

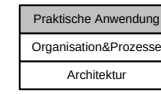
Kapitel 5: Transformation

Der Übergang von einem Modell in ein anderes wird in der MDA *Transformation* genannt. Dieses Kapitel schafft die Grundlagen zu diesem Themengebiet, die erst die Beschäftigung mit der Erstellung von Werkzeugen zur automatisierten Transformation sowie der Generierung von Quellcode als spezielle Unterart ermöglichen.

Kapitel 6: Koordination und Kombination

Nach all den „harten Fakten“ spielen hier die „weichen“ Aspekte der Einführung der MDA in bestehende Unternehmen eine Rolle. Die Integration des Ansatzes in bereits bestehende Software-Prozesse wird hier ebenso beleuchtet wie die Auswirkung des Ansatzes auf die Projektleiter und Entwickler bzw. deren bestehende Aufgaben und Rollen. Weiterhin werden die Anforderungen beleuchtet, die sich an die Unternehmensorganisation ergeben und die speziellen Aufgaben in der Einführungsphase (der MDA) näher betrachtet.

Kapitel 7–9: Vorstellung und Durchführung des Fallbeispiels



Anhand eines Fallbeispiels werden hier die im ersten Teil des Buches erarbeiteten Konzepte dem Lackmustest unterzogen. Konkrete

mögliche Lösungen für technische und konzeptionelle Probleme werden angegeben und auszugsweise detailliert erläutert.

Kapitel 10: /lost+found

Hier werden die Aspekte aufgegriffen, die wir Ihnen nicht vorenthalten wollten, jedoch „nirgendwo sonst so richtig gepasst haben". Weniger ein in sich geschlossenes Kapitel als eine Fundgrube interessanter Ideen und Gedankengänge finden sich hier Aspekte, die das Gesamtbild MDA weiter schärfen.

Kapitel 11: Ende gut – alles gut?

Tief durchatmend werden hier die im Laufe der vielen Kapitel erworbenen Erkenntnisse noch einmal zusammengefasst. Dabei werden wir auch einen vorsichtigen Ausblick in die Zukunft der Softwareentwicklung wagen und auch die Möglichkeit und deren Ursachen für den Fall betrachten, dass sich die MDA *nicht* durchsetzt. Natürlich darf an dieser Stelle auch ein ordentliches Schlusswort nicht fehlen.

Anhang A: UML Schnellreferenz

Im Sinne einer Referenz werden hier alle dreizehn Diagrammarten der UML 2.x aufgeführt und die wichtigsten Notationselemente vorgestellt und erläutert.

Anhang B: OOGPM

Ein Mittel, die Welt der Geschäftsprozessmodellierung mit der MDA zu verbinden, stellt die OOGPM (Objektorientierte Geschäftsprozessmodellierung, [OWS+03]) dar. Hierzu definiert sie sowohl die Methode als auch die notwendigen Profile, um die Ergebnisse als UML Modelle festhalten zu können. An dieser Stelle wird die OOGPM kurz vorgestellt und die wichtigsten Ergebnistypen samt entsprechender Beispieldiagramme erläutert.

1.4 Konventionen

Bei der Gestaltung des Buches wurden Konventionen bei der Wahl der Schrifttypen und der Verwendung von Sprachmitteln verwendet, die an dieser Stelle kurz erläutert werden:

- **Deutsch/Englisch**: Grundsätzlich wurde die im Alltag der Autoren gängige Form gewählt (also etwa *Bridge* und nicht *Brücke*). Wo erforderlich wurden diese in die entsprechende

Form der anderen Sprache übersetzt und in Klammern an den Originalbegriff angehängt. Oftmals wurden der Vollständigkeit halber beide Formen angegeben, um dem Leser eine Orientierung in den Literaturangaben, zu ermöglichen.

- **Codebeispiele**: Codebeispiele und genereller alle textlichen Artefakte, wie Auszüge aus Deskriptoren etc., sind in `Courier New` angegeben (Beispiel: Klasse `Person`).
- **Schlüsselwörter**: Begriffe, die im jeweiligen Kontext eine prominente Bedeutung besitzen sowie Schlüsselkonzepte wurden zur Hervorhebung in *Kursivschrift* gesetzt.
- **Literaturverzeichnis:** Zur besseren Übersichtlichkeit wurden neben dem Gesamtverzeichnis am Ende des Buches die kapitelrelevanten Einträge auch am Ende jedes Kapitels aufgeführt.
- **Index:** Zum besseren Auffinden relevanter Textstellen bei Benutzung des Buches als Referenz und Nachschlagewerk wurde ein ausführlicher Index erstellt, der am Ende des Buches zu finden ist und alle relevanten Schlüsselbegriffe enthält.
- **Marginalien:** Zur Erleichterung der Orientierung innerhalb der einzelnen Kapitel und Abschnitte wurden Leitbegriffe als Marginalie an den Rand der Seite aufgenommen.

1.5 Weitere Informationen

Weitere Informationen, Errata, Quelltexte sowie ein Diskussionsforum zum Austausch untereinander und direkt mit uns, finden sich unter:

```
www.mda-buch.info
```

Literatur

[Eclipse] Eclipse.org: *Eclipse.org Main Page*. http://www.eclipse.org/ (letzter Abruf Mai 2005)

[OWS+03] Bernd Oesterreich, Christian Weiss, Claudia Schröder, Tim Weilkiens und Alexander Lenhard: *Objektorientierte Geschäftsprozessmodellierung mit der UML*. dpunkt, 2003.

2 MDA – Überblick und Orientierung

„Als es noch keine Computer gab, war das Programmieren noch relativ einfach."

Edsger Wybe Dijkstra

In diesem Kapitel werden wir uns mit den Prinzipien der Model-Driven Architecture (MDA) vertraut machen sowie Grundlagen und Basiskonzepte einführen, die eine grobe Einordnung der Konzepte und Techniken ermöglichen und dann in späteren Kapiteln detaillierter beschrieben werden.

Zu Beginn wollen wir aber einen kleinen Blick zurück in die Vergangenheit werfen – auf mittlerweile 50 Jahre industrielle Software-Entwicklung. Dies soll uns dabei helfen, die Motive der Object Management Group (OMG) zu verstehen, die schließlich zur Formulierung des Model-Driven Architecture Ansatzes geführt haben.

2.1 Motivation modellgetriebener Ansätze

Software hat sich im Laufe der letzten Jahrzehnte von ihrem ursprünglichen Nischendasein zur nunmehr kritischen Rolle innerhalb von Unternehmen entwickelt. Viele betriebswirtschaftliche Basisoperationen laufen heute softwaretechnisch automatisiert oder zumindest IT-gestützt ab. Software ist damit zu einem wichtigen Erfolgsfaktor und mitunter zu einem entscheidenden Wettbewerbsvorteil vieler Unternehmen geworden (vgl. [SF03]).

Dem daraus resultierenden steigenden Bedarf an softwaretechnischen Lösungen in der betriebswirtschaftlichen Praxis stehen zunehmend gegenüber:

- eine immer schwieriger zu beherrschende Komplexität,
- stetig steigende Anforderungen an die Leistungsfähigkeit, Zuverlässigkeit und Qualität,
- kurze Technologiezyklen, die sowohl die Hardware- als auch Software-Plattformen betreffen,
- häufige Anforderungsänderungen,
- und (insbesondere in Zeiten konjunktureller Schwächeperioden) ein hoher Druck zur Kostenreduzierung.

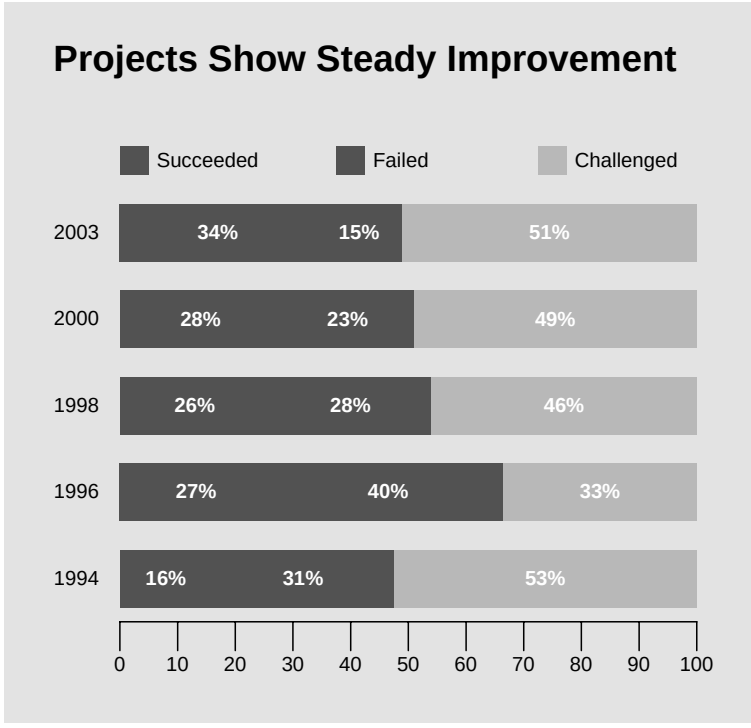


Abb. 2.1 Software-Projekte zeigen stetige Verbesserung (Quelle: [JBCR01] ergänzt durch [SG03])

Spätestens mit dem „Erkennen“ der so genannten Software-Krise [NRB76] wurde daher die Forderung nach ingenieurmäßiger Software-Erstellung immer lauter. Trotz aller Erfahrungen, neuer Technologien und vielfältiger Forschung fehlen der Software-Industrie auch heute noch Entwicklungsprozesse, die an die Planbarkeit, Zuverlässigkeit, Effektivität, Effizienz und Flexibilität der „althergebrachten“ Industrie auch nur annähernd herankommen.

Knowledge-Work vs. Industrielle Fertigung

Ein Versuch, unreflektiert Prozesse aus der Welt der klassischen Industrie in die Software-Welt zu übernehmen, scheitert häufig auf Grund der grundsätzlichen Verschiedenheit der materiellen Fertigung und der sich immer neu stellenden Anforderungen an die Entwicklung „geistiger" Lösungen. Die große Anzahl abgebrochener oder weit über dem ursprünglich angenommenen Budget beendeter Projekte in der Software-Industrie zeigt das.

So hat sich der relative Anteil erfolgreich durchgeführter Software-Projekte laut Studien der Standish Group von 16% im Jahre 1994 auf 34% im Jahre 2003 zwar immerhin mehr als verdoppelt [SG03], andererseits bedeutet dies aber auch, dass immer noch zwei Drittel der Projekte nicht wie geplant beendet werden (siehe auch Abb. 2.1). Projekt-Beispiele der jüngeren Vergangenheit wie *Cheops* [COWO00], *Fiscus* [HO04] oder etwa *Toll Collect* [CT03] bilden hier nur die Spitze des Eisbergs.

Immerhin lässt sich aber ein Fortschritt feststellen. Wir wollen im Folgenden einmal Revue passieren lassen, welche Anstrengungen zu diesen Erfolgen geführt haben.

2.1.1 Die Geschichte der Softwareentwicklung – ein historischer Abriss

Maschinen-sprachen

Beginnen möchten wir in den 50er-Jahren: Zu dieser Zeit bestimmten die knappen und teuren Hardware-Ressourcen die Computer-Welt. Die Erstellung von Software spielte deshalb gegenüber der Hardware-Entwicklung eine untergeordnete Rolle und erfolgte quasi „nebenbei". Software hatte noch eine beherrschbare Größe und wurde in Maschinensprache binär kodiert.

Assembler-Sprachen

Die Software-Systeme wurden komplexer und schon bald musste man erkennen, dass die für den Menschen schwer verständlichen maschinensprachlichen Binärcodes für Befehle und Operanden nicht weiter dazu geeignet waren, um Software zu entwickeln. Die Maschinensprachen wurden durch die Assembler-Sprachen (kurz Assembler) abgelöst, die mnemonische Symbole, Marken und Makros als Arbeitserleichterung für den Entwickler brachte. Mnemonische Symbole lösten die angesprochenen Binärcodierungen ab, Marken und Makros ermöglichten die Abstraktion von einzelnen Instruktionen. Der Assemblierer übersetzte den in Assembler-Sprache geschriebenen Code in ein Maschinenprogramm.

Problemorientierte/Höhere Programmiersprachen

Die Assembler-Sprachen waren nach wie vor maschinenorientiert, gefragt waren aber zunehmend Programmiersprachen, mit denen man die zu behandelnden Probleme direkter formulieren konnte. Die Programmiersprache FORTRAN (FORmula TRANslator), die 1954-1958 von Jim Backus entwickelt worden war, gilt als erste problemorientierte bzw. höhere Programmiersprache. Sie wurde für mathematisch-technische Probleme konzipiert und ist Vorläuferin vieler weiterer höherer Programmiersprachen die folgen sollten. Musste man sich bei der Assembler-Programmierung um Prozessorregister, Stack und Statusflags weitgehend selbst kümmern, konnte man mit den höheren Programmiersprachen von diesen Dingen abstrahieren. Ein Compiler erledigte stattdessen diese Arbeit und sorgte für die Übersetzung in ein ausführbares Programm.

Software-Krise und Software-Engineering

Die Hardware-Preise fielen, der Bedarf an Software wuchs, und schon bald wurden die programmierten Systeme größenmäßig so komplex, dass sie unhandhabbar wurden. Auf einer NATO-Konferenz im Jahre 1968 reagierte man auf diesen Missstand – die Software-Krise wurde ausgerufen und mit ihr wurde die Forderung nach einer angemessenen Ingenieursdisziplin immer vehementer. Bis zu diesem Zeitpunkt war die Software-Erstellung eher als künstlerische Tätigkeit aufgefasst worden, nun sollte eine strukturierte Vorgehensweise die „Kunst“ verdrängen. In dieser Zeit wurde der Begriff des Software-Engineering geprägt (vgl. [Bau93] bzw. [Boe76]).

Strukturierte Programmierung

Mit komfortablen Kontrollstruktur- und Fallunterscheidungsmöglichkeiten begann man nun strukturiert – etwa in Pascal oder in C – zu programmieren [DDH72]. Die neue Disziplin Software-Engineering hatte sich aber nicht nur strukturierte Programmierung auf die Fahnen geschrieben, sondern insbesondere auch systematische Methoden. Diese Lücke wurde unter anderem durch die Strukturierte Analyse (SA) [DeM78] und das Strukturierte Design (SD) [YC79] geschlossen.

Strukturierte Analyse und Strukturiertes Design

In der *Strukturierten Analyse* verwendete man hierarchisch angeordnete Datenflussdiagramme zur abstrakten Modellierung von Prozessen. Mini-Spezifikationen (z. B. in Pseudo-Code) dienten dabei zur Beschreibung von nicht weiter verfeinerten Prozessen und Data Dictionaries stellten die einheitliche Datendefinition sicher. Die in der Strukturierten Analyse ausgemachten Funktionen wurden dann im Strukturiertem Design in hierarchisch aufgebaute Module zerlegt und in Strukturdiagrammen festgehalten. Hierbei dienten die Diagramme der Visualisierung von Programmabläufen und der Be-

schreibung der Schnittstellen zwischen den Modulen. Diagramme bzw. Modelle erhielten in den strukturierten Methoden einen hohen Stellenwert und ein Markt für Modellierungs-Werkzeuge entstand: Willkommen in der Welt des Computer Aided Software Engineering (CASE)!

Wiederverwendungskrise

Die *Wiederverwendungskrise* [Qui94] gegen Ende der 80er-Jahre machte diese Idylle zunichte. Die abermals drastisch in ihrer Komplexität gewachsenen Software-Systeme erzwangen erneut ein Umdenken bei der Software-Erstellung. Diesmal wollte man insbesondere der Wiederverwendbarkeit von Software Rechnung tragen. So hatte man zwar im Laufe der Jahre gewaltige Software-Bestände angehäuft, aber gleichzeitig das Problem, diese in geeigneter Form auch nur teilweise wieder zu verwenden.

Objektorientiertes Paradigma

Die Lösung sollte das Objektorientierte Paradigma liefern; ein Ansatz, der der natürlichen Denkweise sehr nahe kommt: Ein System besteht aus vielen Objekten und jedes dieser Objekte besitzt ein definiertes Verhalten, einen inneren Zustand und eine eindeutige Identität. Bereits in den 70er-Jahren hatte man sich diesem Ansatz in wissenschaftlichen Veröffentlichungen gewidmet, der Durchbruch sollte erst jetzt erfolgen.

Ole-Johan Dahl und Kristen Nygaard hatten bereits in den 60er-Jahren eine entsprechende Programmiersprache entwickelt: SIMULA (SIMUlation LAnguage). Ursprünglich für diskrete Ereignis-Simulation entwickelt, kamen ihre Konzepte so zu spätem Ruhm – die breite Akzeptanz wiederum, sollte später aber Sprachen wie Smalltalk, C++ und Java vorbehalten bleiben.

Daten und Funktion wurden fortan nicht mehr getrennt voneinander betrachtet, sondern zusammenhängend in Objekten. Klassen nahmen dabei die Objekte auf, die dem gleichen Verhaltensschema folgten und die gleiche innere Struktur besaßen. Vererbung und Polymorphie sollten für die oben angesprochene Wiederverwendbarkeit Sorge tragen. Um die Übersetzung des Programm-Codes in ein ausführbares Programm kümmerte sich weiterhin ein – um OO-Konstrukte aufgebohrter – Compiler.

Objektorientierte Analyse und Objektorientiertes Design

Der breiten Akzeptanz folgend schossen Anfang der 90er-Jahre adäquate objektorientierte Methoden wie Pilze aus dem Boden - Rumbaughs Object Modeling Technique (OMT) [RBP+91] und Boochs Object-Oriented Design (OOD) [Boo93] waren bzw. sind wohl die zwei Bekanntesten unter ihnen. Sie intensivierten abermals den Gebrauch von Modellen und entsprechender Modellierungs-Werkzeuge in der Software-Entwicklung. Aus der Vielzahl der

objektorientierten Methoden ging die Unified Modeling Language (UML) hervor, eine standardisierte Modellierungssprache zur Spezifikation objektorientierter Systeme.

Middleware, Komponenten und Applikationsserver

In jüngster Vergangenheit bekam der Entwickler Unterstützung durch eine Reihe verschiedener Konzepte wie Middleware [MW], Komponenten und Applikationsservern. Durch ihre Verbreitung wurden die Rechnergrenzen endgültig gesprengt und mit ihr eine Transparenz der Netzwerkkommunikation geschaffen. Komponenten brachten das Paradigma des Zusammenbaus vorgefertigter Software-Einheiten. Die beiden Konzepte wurden mithilfe von Object Request Brokern (ORBs) bzw. Applikationsservern zur verteilten Objekttechnologie bzw. Distributed Object Technology (DOT) verheiratet. Zusammengenommen verschafften sie uns insbesondere Ansätze zur Abstraktion von Verteilung, Persistenz, Transaktionalität und Sicherheit.

2.1.2 Die Gegenwart

Damit sind wir am Ende unserer kleinen Rückblicks angekommen. Gegenwärtig wird das Objektorientierte Paradigma auf breiter Basis erfolgreich um- und eingesetzt und ist aus der Entwicklungspraxis nicht mehr wegzudenken. Mittlerweile sehen wir uns hochkomplexen Zielarchitekturen gegenüber, müssen die „Jugendsünden“ in Form bestehender Anwendungen integrieren und dabei die verschiedenartigsten Technologien einbeziehen. Auch hierbei unterstützen uns Modelle und das damit verbundene Prinzip der Abstraktion, indem sie helfen, komplexe Sachverhalte kompakt darzustellen.

Welche Schlussfolgerung lässt sich nun ziehen?

Betrachtet man die eben beschriebene Entwicklung der Software-Entwicklung unter dem Gesichtspunkt eines gemeinsamen „Trends“ fallen folgende Punkte besonders ins Auge:

*Komplexitätsbeherrschung als **das** Problem*

- Der Fortschritt in der Software-Entwicklung ist eng verbunden mit der Bewältigung von Komplexität – mitunter spricht man deshalb bei der Komplexitätsbeherrschung auch von *dem* Problem des Software-Engineering.

Abstraktion als fortwährender Trend

- Zu beobachten ist weiterhin, dass immer, wenn die Komplexität der zu erstellenden Software-Systeme auf ein nicht mehr zu beherrschendes Maß angewachsen war, die Lösung in der Vergangenheit in einem Abstraktionssprung auf eine höhere semantische Ebene zu finden war.

Diese Beobachtungen lassen sich auch an der Evolution der Programmiersprachen festmachen: von Maschinensprachen über Assembler-Sprachen, Prozeduralen Sprachen hin zu Objektorientierten Sprachen. Insofern kann bei der Abstraktion von einem fortwährenden Trend des Software-Engineering bzw. allgemein der Informatik gesprochen werden.

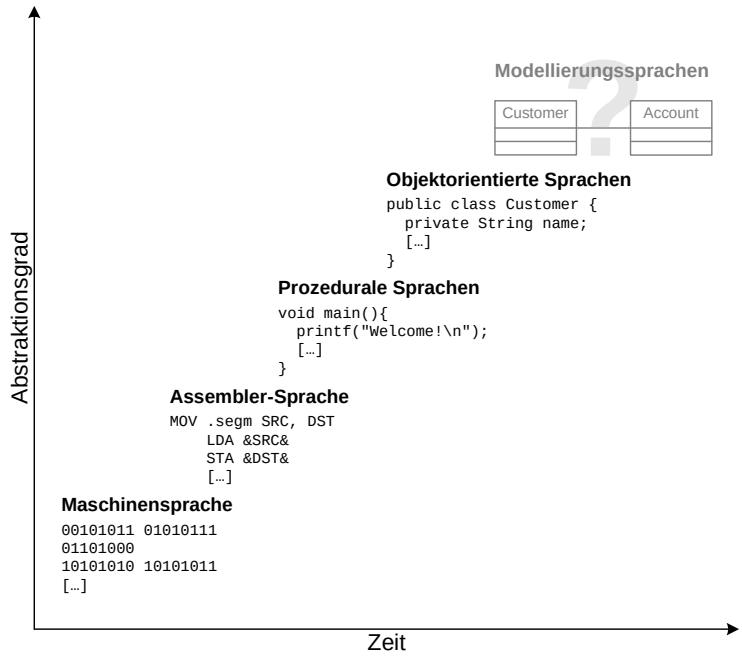


Abb. 2.2 Steigender Abstraktionsgrad

Wie bereits erwähnt sind Modelle ein weiteres Werkzeug, das uns bei der Beherrschung dieser Komplexität hilft, indem sie frei nach dem Motto *„teile und herrsche“* einen Teil der selbigen verdecken. Diese Beobachtung liefert den dritten Punkt, den wir als Trend in der Historie der betrachteten Entwicklung festmachen wollen:

- Modelle rücken mehr und mehr in den Mittelpunkt des Entwicklungsprozesses, insbesondere in Analyse und Entwurf.

Zunehmende Gewichtung von Modellen

Wie geht es weiter?

Angenommen der ausgemachte Trend wäre eine Tatsache, wäre es dann nicht konsequent zu folgern, dass Modellierungssprachen die

Programmiersprachen der Zukunft sind? Abb. 2.2 veranschaulicht diese Idee.

Nun, die Folgerung dieser Konsequenz aus dem bloßen Wissen der geschichtlichen Entwicklung und den damit verbundenen subjektiven Tendenzwahrnehmungen abzuleiten erscheint uns an dieser Stelle dann doch etwas gewagt.

2.1.3 Akute Probleme bei der Software-Erstellung

Aber schauen wir uns doch einmal die derzeitigen Probleme bei der Software-Erstellung einmal genauer an – vielleicht können wir einen Zusammenhang in ihnen ausmachen und aus diesem heraus unsere These weiter untermauern. Unter anderem trifft man häufig auf die folgenden Probleme (vgl. [You02], [Gla02] und [SPL03]):

Dominierung von Fachlichkeit durch Technik

- Die Umsetzung der fachlichen Basiskonzepte wird immer noch von der Technik dominiert. Anwendungsentwickler müssen umfangreiches technisches Wissen mit- bzw. einbringen, statt sich auf die fachliche Anwendungsdomäne und damit auf ihr fachliches Anwendungswissen konzentrieren zu können. Die Lücke zwischen fachlichem Problemraum und technischer Lösung klafft nach wie vor: Nicht nur, dass sich Fachabteilung und Entwickler in verschiedenen Sprachen zu verständigen versuchen, sie tun dies auch immer noch auf völlig unterschiedlichen Abstraktionsebenen. Es hat sich damit gezeigt, dass das Abstraktionsniveau der derzeitigen Entwicklungsansätze immer noch zu niedrig ist, der Traum von einer rein durch die Fachlichkeit bestimmten Entwicklung scheint immer noch in weiter Ferne.

Divergenz der Änderungs-zyklen

- Zum einen ist Technologie einem fortwährenden und hochfrequenten Änderungszyklus unterworfen, während sich Großteile der fachlichen Basis vergleichsweise langsam wandeln. Zum anderen sind die fachlichen Prozesse an der Speerspitze der Unternehmen der sich im ständigen Fluss befindlichen Gesetzeslage respektive den aktuell herrschenden Marktbedingungen unterworfen (Stichwort: Globalisierung/Deregulierung).
  Um einen besseren Investitionsschutz der Entwicklungsarbeit zu gewährleisten und eine verlängerte Lebensdauer der Fachkonzepte erreichen zu können, gleichzeitig aber auch eine größere Flexibilität und bessere Reaktionsfähigkeit bei sich ändernden Fachanforderungen zu erlangen, wäre es sinnvoll, technische und fachliche Belange der Entwicklung zu trennen, um damit

eine Entkopplung der Änderungszyklen von Fachlichkeit und Technik zu erhalten.

- *Methodischer Bruch zwischen Analyse, Design und Implementierung*

  Beim Übergang von modellbasierter Analyse und modellbasiertem Entwurf zur programmiersprachlichen Implementierung entsteht ein methodischer Bruch. Die Modelle eines Software-Projektes (=ein wichtiger Teil der Dokumentation) sind dadurch einem erheblichen Alterungs- bzw. Verfallsprozess unterlegen. Viele Modelle sind also schon kurz nach Beginn der Implementierungsphase durch erste nicht nachdokumentierte Änderungen nicht mehr aktuell und damit nahezu nutzlos. Auch Änderungswünsche des Kunden sowie das während der Implementierungsphase fortschreitende Verständnis der fachlichen und technischen Anforderungen, verlangen Architekturanpassungen des Software-Systems, die oft nicht in der Dokumentation nachgezogen werden. Gerade die Einarbeitung neuer Mitarbeiter in späten Phasen eines Projektes wird so nicht effizient durch die vorhandene Dokumentation unterstützt. Darüber hinaus erhöhen sich die Rüstzeiten in Betriebs- bzw. Wartungsphase durch die fehlende, mangelhafte bzw. ungültige Dokumentation um ein Vielfaches.

- *Fehlende Traceability*

  Auch wenn sich der Bruch bei dem gerade geschilderten Übergang aus Entwicklersicht am deutlichsten zeigt, ist eine fehlende Durchgängigkeit für den kompletten Lebenszyklus einer Software festzustellen, also von ihrer initialen Definition über Anforderungsanalyse, Realisierung bis hin zu ihrer Stilllegung oder Modernisierung. Im Sinne des Requirements Engineering ist hier zu vorderst die fehlende *Nachvollziehbarkeit bzw. Rückverfolgbarkeit (engl. Traceability)* von Anforderungen zu beklagen.

- *Äußere und innere Gleichförmigkeit von Projekten*

  Zwar impliziert ein Projekt immer einen neuartigen und einmaligen Charakter, oftmals werden aber mehrere Projekte in der gleichen Fach- und/oder Technikdomäne durchgeführt. Genau so oft ähneln oder gleichen sich daher die (architekturellen) Konzepte der realisierten Anwendungen. Diese Gleichförmigkeit wird häufig übersehen oder schlicht ignoriert und anstatt diese domänen-spezifischen Konzepte wiederverwendbar zu machen, wird die gleiche Arbeit unnötigerweise mehrfach verrichtet.

  Neben dieser äußeren Gleichförmigkeit von Projekt zu Projekt findet man immer wieder innere Gleichförmigkeiten innerhalb eines Projektes vor. In diesem Zusammenhang spricht man auch vom *Copy+Paste+Modify-Phänomen*, da die ähnlichen Anteile kopiert, eingefügt und dann geringfügig modifiziert werden. Ein Ansatz zur geeigneten Herausfaktorisierung sowohl der Be-

standteile der äußeren als auch der inneren Gemeinsamkeiten wäre wünschenswert.

Descriptor Hell

- Das Konzept der Middleware hat zwar weitgehend von den technischen Aspekten der Verteilung abstrahiert, die Verteilungsflexibilität wurde aber mit dem Zwang zu umfangreicher Konfiguration erkauft. Zum einen könnte ein Teil der Einstellungen durch sinnvolle Vorgabewerte abgedeckt werden, zum anderen legt ihre deklarative Beschaffenheit eine Umsetzung in Diagrammform nahe. Als Beispiel seien hier Einstellungen zu Sicherheit, Lastverteilung und Transaktionalität genannt. In ihrer derzeitigen Form macht die inhärente Komplexität dieser Einstellungen es mühsam und zeitaufwendig zu ersten diskutierbaren Ergebnissen zu gelangen.

Middleware Babel

- Die Verwendung einer Middleware wird erschwert, wenn man sich außerhalb eines homogenen Umfelds begibt und sich nicht mehr auf genau eine Middleware stützen kann. Die Praxis ist durch heterogene Systemlandschaften geprägt, in denen sich geradezu ein Babel verschiedener Middleware-Technologien auftut. Die Sicht auf die zu integrierenden Systeme wird durch diesen Umstand verschleiert und der Integrationsprozess gestaltet sich zunehmend schwierig.

Legacy Crisis

- Mit der zunehmenden Verkürzung der technologischen Änderungszyklen wächst auch der Druck zur Modernisierung bereits bestehender Anwendungen. Seit 1970 hat sich der Anteil der Kosten für Wartung und Modernisierung bestehender Software, gemessen am Gesamtbudget, von ehemals 40 Prozent auf heutzutage über 90 Prozent gesteigert, sodass immer weniger Geld für die Neuentwicklung von Software zur Verfügung steht [Erl03]. Dieses als *Legacy Crisis* bekannt gewordene Phänomen hat seine Ursachen vor allem in der fortschreitenden Degeneration von Software-Systemen ab dem Zeitpunkt ihrer Fertigstellung. Abb. 2.3 aus [SPL03] verdeutlicht die Ursachen hierfür anschaulich.

  Anwendungen werden gebaut, um fachliche Anforderungen zu erfüllen. Mit der Zeit kommen neue Wünsche der Stakeholder hinzu, die eine Lücke zwischen gewünschter und gelieferter Funktionalität (die gestrichelten Linien) entstehen lassen – repräsentiert durch den wachsenden grauen Bereich im Diagramm. Durch die fortwährende Wartung des Systems wird zwar versucht, diese Lücke wieder zu schließen, was auch annäherungsweise gelingt, durch die ständigen Anpassungen wird jedoch gleichzeitig die unterliegende Struktur der Anwendung immer komplexer und unwartbarer (man spricht hier von der

Degeneration der Anwendungsarchitektur).

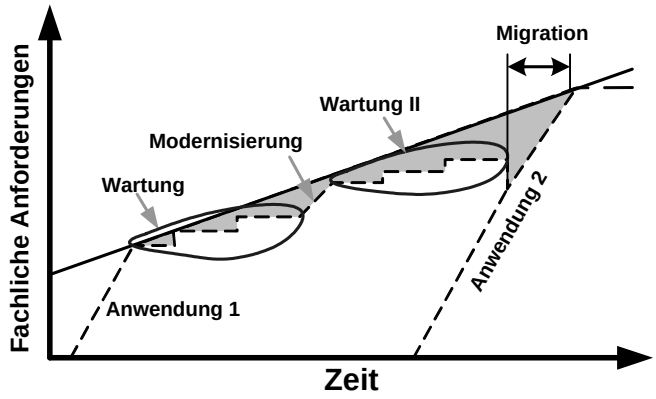


Abb. 2.3 Lebenszyklus von Anwendungen [SPL03]

Die Folge ist schließlich ein Rückstau der fachlichen Änderungswünsche, die Lücke wird immer größer. Schließlich wird die Divergenz so groß, dass eine Modernisierung der Anwendung erforderlich wird. Nach weiteren Phasen der Abwechslung von Wartung und Modernisierung erreicht die Systemkomplexität einen Grad, der die Neuentwicklung der kompletten Anwendung notwendig macht und zusätzliche Kosten durch die Folgen der sich anschließenden Migration verursacht. Ist auch diese vollzogen, beginnt der beschriebene Zyklus von vorn, ein Ausbrechen aus diesem Kreislauf ist also unbedingt notwendig.

Aus den geschilderten Problemen lässt sich das Folgende ablesen: Es besteht der Bedarf, sich auf einem höheren Abstraktionsniveau in domänen-spezifischer Weise ausdrücken zu können. Gleichzeitig besteht das Potenzial, Gleichförmigkeiten in einer verdichteten Form zusammenzufassen. Darüber hinaus gibt es die Forderung nach einer durchgängigeren Auswahl der Ausdrucksmittel.

Models to the rescue!?

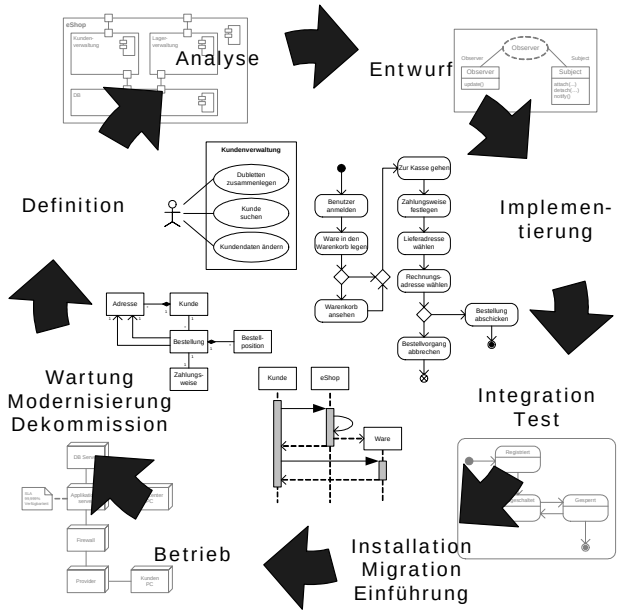
Modelle haben eine gewisse Tradition in dieser Problematik zu helfen: Sie abstrahieren und fokussieren auf das Wesentliche und sie schlagen eine Brücke von der fachlichen Problemwelt in die technische Lösungswelt.

2.1.4 Die Idee modellgetriebener Ansätze

Die bisher gemachten Beobachtungen legen also den Schluss nahe, Modellen mit ihrer kompakten Darstellung und ihrer hohen Ausdrucksstärke eine umfassendere Bedeutung als bisher innerhalb des

Entwicklungsprozesses zu geben. Zukünftig sollen sie dem kompletten Lebenszyklus einer Software dienen (siehe Abb. 2.4).

Abb. 2.4 Modelle im Mittelpunkt des Software-Lebenszyklus



Modell-getriebener Ansatz

In einem *modellgetriebenen Ansatz* werden Modelle zu Artefakten erster Klasse, hinreichend zur Generierung eines Software-Systems. Dies kommt einem Übergang von *deskriptiver* (=beschreibender) zu *präskriptiver* (=vorschreibender) Verwendung von Modellen im Software-Entwicklungsprozess gleich.

Gleichzeitig müssen wir nach all der mühevoll erworbenen Erfahrung feststellen: Es kann nicht die *eine* universelle Entwicklungssprache geben, die alle Problemdomänen adäquat widerspiegeln kann. Für den Schritt auf die nächsthöhere Abstraktionsstufe (vgl. Abb. 2.2) ist die *eine* Sprache, die lediglich eine dichtere Notation für eine Teilmenge von Konzepten gegenüber klassischen Sprachen bietet, allein nicht ausreichend. Vielmehr benötigen wir die Möglichkeit, problemorientierte oder besser – weil dieser Begriff vorbelastet ist – *domänen-spezifische Sprachen* selbst definieren zu können und zwar so, dass wir ihnen semantische Eindeutigkeit verleihen können. Im Grunde wird der Entwickler durch diesen Schritt zum Compilerbauer.

Ein Ansatz, der diese Ideen umsetzten möchte, ist die Model-Driven Architecture (MDA) der Object Management Group (OMG). Die

MDA ist dabei als Spezifikation für eine Architektur zur Durchführung eines solchen Entwicklungsansatzes zu verstehen. Mit *Architektur* ist jedoch *nicht* die Architektur konkreter Software-Anwendungen gemeint, sondern vielmehr die notwendige Infrastruktur zur Durchführung eines MDA-basierten Ansatzes vom unterliegenden Software-Prozess bis hin zu den verwendeten Hilfsmitteln und Werkzeugen.

2.2 Die Model-Driven Architecture (MDA)

Kernidee der MDA

Der Ansatz folgt der Grundidee, dass die Spezifikation einer Softwarekomponente unabhängig von ihrer technischen Umsetzung beschrieben werden sollte. Die MDA bedient sich dabei der Unterscheidung verschiedener Abstraktionslevel, auf denen sich Modelle befinden können und nennt namentlich vor allem die *plattformunabhängigen (engl. platform independent)* als konzeptionelle Basis des Ansatzes sowie die *plattform-spezifischen (engl. platform specific) Modelle*. Der Begriff der Plattform muss dabei relativ aufgefasst werden: die spezifischen Modelle der einen Ebene können die plattform-unabhängige Vorlage für die Modelle der nächsten Stufe sein und umgekehrt.

Weiterer Kerngedanke: Der Übergang von abstrakteren hin zu technologiespezifischeren Modellen erfolgt vorzugsweise vollautomatisiert durch die Verwendung von Transformationswerkzeugen. Die Beschreibung der Transformationen findet dabei in gesondert vorzuhaltenden Transformationsbeschreibungen statt.

2.2.1 Ziele der MDA

Die Ziele, die mit diesem Vorgehen verknüpft sind, lassen sich grob in die folgenden Kategorien einordnen:

- **Konservierung der Fachlichkeit:** Wie bereits als Problem erkannt, ist wertvolles Wissen über die Kerngeschäftsprozesse eines Unternehmens oft implizit in den existierenden proprietären Altsystemen bzw. in den Köpfen der Entwickler verborgen. Durch eine plattform-unabhängige Modellierung der unterliegenden Prozesse kann es explizit gemacht werden und so anschließend als Ausgangspunkt der Pflege und Weiterentwicklung der Anwendungslogik sowie zur Dokumentation und Evolution der Prozesse selbst dienen. Durch die Integration der so entstandenen dokumentierenden Modelle in die Transformationskette hin zur Anwendung wird zudem dem Problem der Degeneration von Dokumentation begegnet.

- **Portierbarkeit:** Durch eine systematische Abstraktion von technischen Aspekten erleichtert MDA deutlich die spätere Migration von Applikationen auf neue Versionen der benutzten Technologien, die Portierung auf komplett andere Zielumgebungen sowie die Anwendungsentwicklung für grundsätzlich mehr als eine Zielplattform.
- **Systemintegration und Interoperabilität:** Durch die konsequente Trennung der fachlichen Konzepte von der konkreten Repräsentation dieser Fachlichkeit in der Technologie existierender Systeme wird die Bildung von Schnittstellen erleichtert, mit denen andere Anwendungen angebunden werden können. Zusammen mit der konsequenten Verwendung offener Standards verringert sich das Risiko eines Vendor-Lock-Ins und hilft dabei, bestehende Anwendungen auch über immer kürzer werdende Technologiezyklen zu retten. Zusätzlich wird die Wiederverwendung bereits bestehender Komponenten erleichtert und dadurch die Produktivität durch Vermeidung unnötiger Redundanzen in der Software-Entwicklung gesteigert.
- **Effiziente Softwareentwicklung:** Durch ein hohes Maß an Werkzeugunterstützung und der damit verbundenen Automatisierung der Anwendungserstellung werden Software-Prozesse beschleunigt, Aufwände verringert und die Qualität der Software, etwa durch die konsequente Verwendung von Referenzarchitekturen und Pattern-Replication gesteigert. Die immer komplexer werdende Anwendung moderner Technologien wird durch geeignete Abstraktion gekapselt und so das Wissen von Technologie-Experten in Form von Transformationsvorschriften multiplizierbar gemacht. Dies ermöglicht die optimale Nutzung vorhandener Ressourcen auf der einen und gleichzeitig die Beherrschung von systemimmanenter Komplexität durch das Prinzip *Teile-und-herrsche* auf der anderen Seite.
- **Domänen-Orientierung:** Mittels Erstellung domänenspezifischer Modelle und der Wiederverwendung entsprechender Architekturmetamodelle wird nicht nur die Erstellung neuer Anwendungen in kürzeren Zeiträumen – und damit die Reduzierung der Time-to-Market ohne Erhöhung des Budgets – ermöglicht, sondern zudem ein Wissensvorsprung weg von der reinen Technikzentrierung innerhalb des wertvolleren fachlichen Domänenwissens gesichert. Die prominente Stellung der Fachlichkeit innerhalb des Software-Entwicklungszyklus wird so dem Bedürfnis der Kunden nach hoher Flexibilität/Agilität der unterliegenden Geschäftsprozesse ohne unnötige Beschränkung durch technologische Vorgaben seitens der IT gerecht.

2.2.2 Die Vorgaben der Object Management Group (OMG)

Die Ausführungen der OMG zu konkreten Vorgehensanweisungen in MDA-Projekten sind auch fünf Jahre nach Veröffentlichung des ersten Whitepapers immer noch sehr vage. Sie besitzen zum Teil eher Konzept-Charakter, als dass sie eine Spezifikation darstellen, die quasi nur darauf wartet, implementiert bzw. umgesetzt zu werden. Nichtsdestotrotz wollen wir uns den raren Vorgaben, die seitens der OMG im Kontext MDA existieren, ausführlich widmen, um uns einen ersten Überblick zu verschaffen.

Die OMG beschreibt in [MDAATP] und [MDAGuide] zunächst einige Basiskonzepte, von denen wir im Folgenden die Wichtigsten vorstellen. Dabei lässt es sich nicht vermeiden, sich vorab durch eine Menge Begriffsbestimmungen zu arbeiten, um sich später in der Flut von Begriffen, die innerhalb der MDA eine zentrale Rolle spielen, nicht zu verlieren und jederzeit den Überblick zu behalten.

Zur Darstellung der MDA in diesem Abschnitt ist noch anzumerken, dass hier in erster Linie die „reine Lehre“ der OMG dargestellt und auf eine Interpretation soweit wie möglich verzichtet wird.

Terminologie IEEE1471

Bei ihren Beschreibungen stützt sich die OMG auf die „Empfohlenen Praktiken zur architekturellen Beschreibung von softwareintensiven Systemen“, die im ANSI/IEEE Standard 1471 [IEEE1471] festgehalten sind. Ein Ausschnitt aus der darin verwendeten Begriffswelt ist in Abb. 2.5 dargestellt. Da die hier definierten Begriffe die Basis bei der standardisierenden Arbeit mit Software-Architekturen darstellen, wollen wir zuerst diese erklären, bevor wir sie mit denen der Begriffswelt der MDA in Bezug setzen.

System

Ein *System* ist ein aus Teilen zusammengesetztes und strukturiertes Ganzes. Es hat eine Funktion, erfüllt einen Zweck und verfügt über eine Architektur.

Architecture

In der *Architektur* finden sich die Organisation der Teile eines Systems, deren Verhalten sowie die Beziehungen unter den einzelnen Komponenten wieder. Beziehungen können zudem an Bedingungen geknüpft sein, die zur Erreichung des Systemzwecks erfüllt werden müssen.

Architectural Description

Eine *architekturelle Beschreibung* besteht aus der Menge aller Artefakte, die zur Beschreibung einer Architektur benötigt werden. Also typischerweise aus Modellen des Systems und unterstützender Dokumentation.

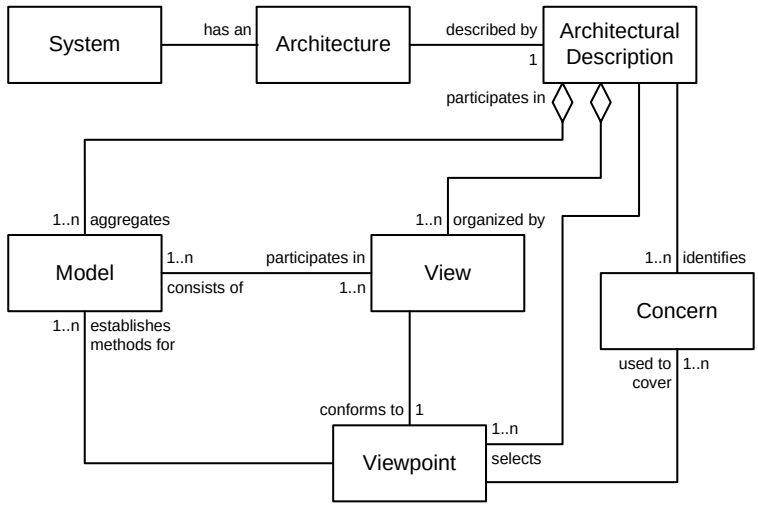


Abb. 2.5 Terminologie des IEEE 1471 Standards

Concern

Concerns (zu Deutsch: Belang, Angelegenheit, Interesse oder Aspekt) sind spezifische Anforderungen oder Bedingungen, die erfüllt werden müssen, um die erfolgreiche Erfüllung des Systemzwecks zu gewährleisten. Im Folgenden übersetzt als *Aspekt*.

Model

Ein *Modell* beschreibt oder spezifiziert ein System zu einem bestimmten Zweck. Dazu erfasst es alle hierfür relevanten Aspekte, etwa Struktur, Verhalten, Funktion und die Umwelt des Systems.

Viewpoint

Viewpoints (dt. Gesichtspunkt, Perspektive) beschreiben Elemente und Konstrukte, mit denen die Erstellung eines Modells zur Beschreibung von Systemaspekten ermöglicht wird; einfacher gesagt: die Regeln, Techniken bzw. Methoden sowie die Notationsmittel zur Erstellung eines Modells.

View

Eine kohärente Menge von Modellen, die bestimmte Aspekte eines Systems beschreibt und die nichtrelevanten verbirgt, wird als *Sicht (engl. View)* auf dieses System bezeichnet. Eine Sicht wird mit den Regeln und der Notation des entsprechenden Viewpoints (s.o.) dargestellt.

2.2.3 Metamodell der zentralen MDA-Begrifflichkeiten

Wenden wir uns nun der MDA-Welt zu. Abb. 2.6 zeigt das Metamodell ihrer zentralen Begrifflichkeiten.

Metamodel

Ein *Metamodell* ist die Menge von Elementen mit denen Modelle, die sich auf dieses Metamodell beziehen, erstellt werden können. Es enthält die Regeln, die bei der Erstellung des Modells beachtet werden müssen (abstrakte Syntax) sowie die Bedeutung der Elemente und Elementkonstellationen (Semantik).

Profile

Ein *Profil* ist eine *leichtgewichtige Erweiterung* eines Metamodells. Dazu stellt es neue Modellelemente zur Verfügung oder redefiniert bzw. erweitert die Semantik oder Syntax bereits bestehender Elemente. Im Gegensatz zur Definition völlig neuer Metamodelle (schwergewichtiger Ansatz) können sie die Bedingungen, die an Elemente der Modelle gestellt werden, nur verschärfen und müssen sich so den Regeln eines Design-by-Contracts [DBC] unterordnen. Kapitel 3 setzt sich näher mit diesem Mechanismus der UML auseinander.

Domain

Unter dem Begriff der *Domäne* versteht man ein abgrenzbares, kohärentes (Wissens-)Gebiet. In einem Metamodell (respektive in einem Profil) werden die Konzepte einer Domäne in Bezug gesetzt und beschrieben. Zusammen mit ihrem sprachdefinierenden Charakter bezeichnet man Metamodell und Profil daher als *Domänenspezifische Sprache (engl. Domain-Specfic Language, DSL).*

Application

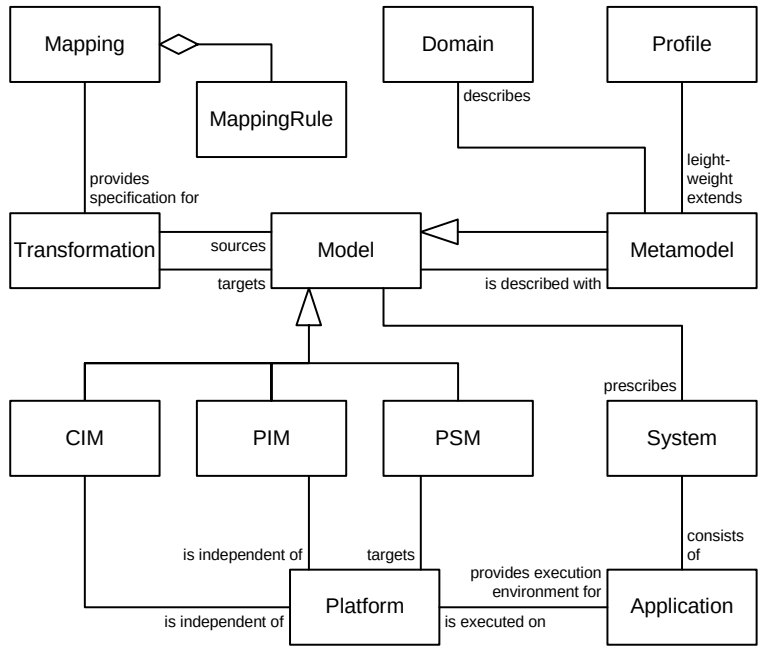
Eine *Anwendung oder Applikation (engl. Application)* ist ein zu erstellendes Stück Software, welches die zu entwickelnde Funktionalität umfasst. Ein System kann sich aus einer oder mehreren

Anwendung(en) zusammensetzen, die auf einer oder mehreren Plattform(en) ausgeführt werden.

Plattform

Eine *Plattform* ist eine Ausführungsumgebung für eine Anwendung. Ihre Funktionalität stellt sie durch zweckmäßige Schnittstellen bereit, auf die die Anwendung ohne Kenntnis der eigentlichen Implementierung zugreifen kann. Typische Beispiele für Plattformen sind im Bereich Betriebssysteme (→Unix, Windows, OS/390), Programmiersprachen (→C++, Java, C#) und Middleware (→CORBA, Java EE, .NET) zu finden. Typisch für Plattformen ist auch, dass sie oft in der Art eines Platform-Stacks (dt. in etwa Plattform Stapel) aufeinander aufsetzen. So bildet die Hardware eines Computers die unterliegende Plattform für das Betriebssystem, das seinerseits wieder Plattform für die einzelnen Anwendungen bietet usw.

Abb. 2.6 MDA Metamodell der zentralen Begrifflichkeiten



Für eine ausführlichere Beschreibung des Plattform-Gedankens sowie der nun folgenden zentralen Modellarten der MDA verweisen wir auf die Ausführungen in Kapitel 4 – Modelle der MDA, der wir an dieser Stelle nicht vorgreifen und daher nur eine essenzielle Definition liefern wollen.

Computation Independent Model (CIM)

Das *berechnungs-unabhängige Modell* liefert eine Sicht auf das Gesamtsystem, die völlig unabhängig davon ist, wie es implementiert wird. Das Modell wird im Vokabular seiner Domäne beschrieben und betont die Anforderungen an das System und seine Umwelt.
Synonyme: Business Model, Domain Model

Platform Independent Model (PIM)

Das *plattform-unabhängige Modell* der MDA beschreibt formal die Struktur und Funktionalität eines Systems. Es ist unabhängig von implementierungstechnischen Details und abstrahiert damit von der zugrunde liegenden Plattform.

Platform Specific Model (PSM)

Durch eine Anreicherung des PIM mit plattform-abhängigen Informationen erhält man das *Platform Specific Model (plattform-spezifisches Modell).* Ein PSM wird zur Implementierung, falls es alle Informationen zur Konstruktion und zum Betrieb eines Systems liefert. Alternativ können in einem solchen Fall die Modelle auch direkt ausgeführt werden, man spricht dann von *ausführbaren Modellen (engl. executable models).*

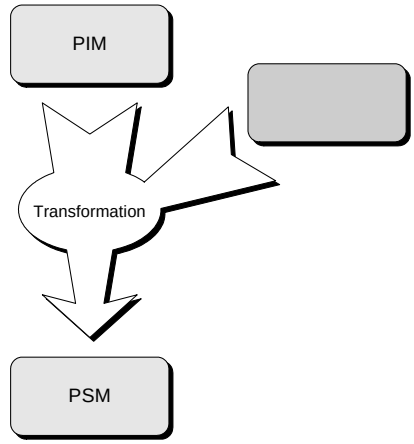


Abb. 2.7 Transformations-muster abstrakt

Transformation

Eine *Transformation* (oder genauer Modell-Transformation) ist der Prozess der Umwandlung eines Modells in ein anderes Modell des gleichen Systems. Die Spezifikation für eine solche Transformation

ist in *Abbildungsregeln (engl. Mapping Rules)* festgehalten, die in einer *Abbildung (engl. Mapping)* gebündelt sind.

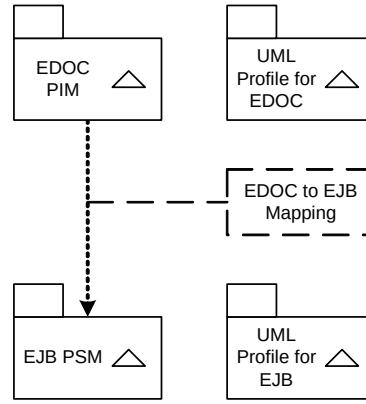
Dieses Muster ist das Kernstück des MDA-Ansatzes. Abb. 2.7 veranschaulicht diesen Prozess für die Transformation eines PIM in ein PSM.

Die Darstellung ist dem MDA-Guide [MDAGuide] entlehnt und dort als „bewusst andeutend und generisch“ gekennzeichnet. Sie lässt nicht darauf schließen wie die Überführung vonstatten geht. Sie veranschaulicht lediglich, dass zur Transformation in ein PSM das PIM selbst und „weitere Informationen“ benötigt werden (leerer Kasten). Wie diese „weiteren Informationen“ im Einzelnen aussehen, werden wir in einem späteren Kapitel sehen (Kapitel 5 – Transformation).

Vorschau auf eine Beispieltransformation

Etwas konkreter werden wollen wir aber an dieser Stelle dann doch und schauen uns – ohne zu sehr vorgreifen zu wollen – einmal an einem einfachen Beispiel an, was uns so eine Transformation liefern kann.

Abb. 2.8 Transformationsmuster konkret



Als Zielplattform wählen wir exemplarisch das Komponentenmodell der Enterprise JavaBeans (EJB) [EJB]. Es ist Teil der Java Enterprise Edition (Java EE) der Firma Sun Microsystems (siehe [JavaEE]). PIM und PSM stellen wir in der Modellierungssprache UML (die Unified Modeling Language, siehe [UML]) dar. Wir werden diese Modellierungssprache im Verlaufe des Buches noch genauer kennen lernen, an dieser Stelle wollen wir uns damit begnügen, dass man mit ihr objektorientierte (Software-)Systeme beschreiben kann.

Das PSM ist im „Vokabular“ des *UML Profile for EJB* [UPfEJB] dargestellt, das PIM in dem des *UML Profile for EDOC* (Enterprise Distributed Object Computing) [UPfEDOC]. Letzteres liefert ein plattform-unabhängiges Metamodell zur Spezifizierung von verteilten komponentenbasierten Software-Systemen. Die Abbildungsregeln für EDOC auf EJB sind Teil dieser Spezifikation. Abb. 2.8 veranschaulicht diesen Zusammenhang.

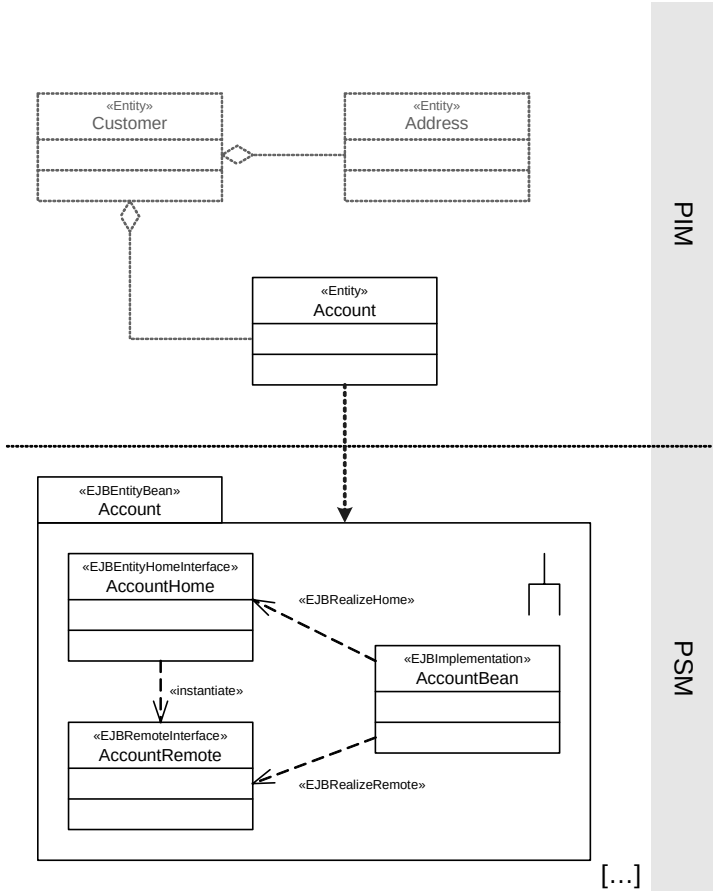
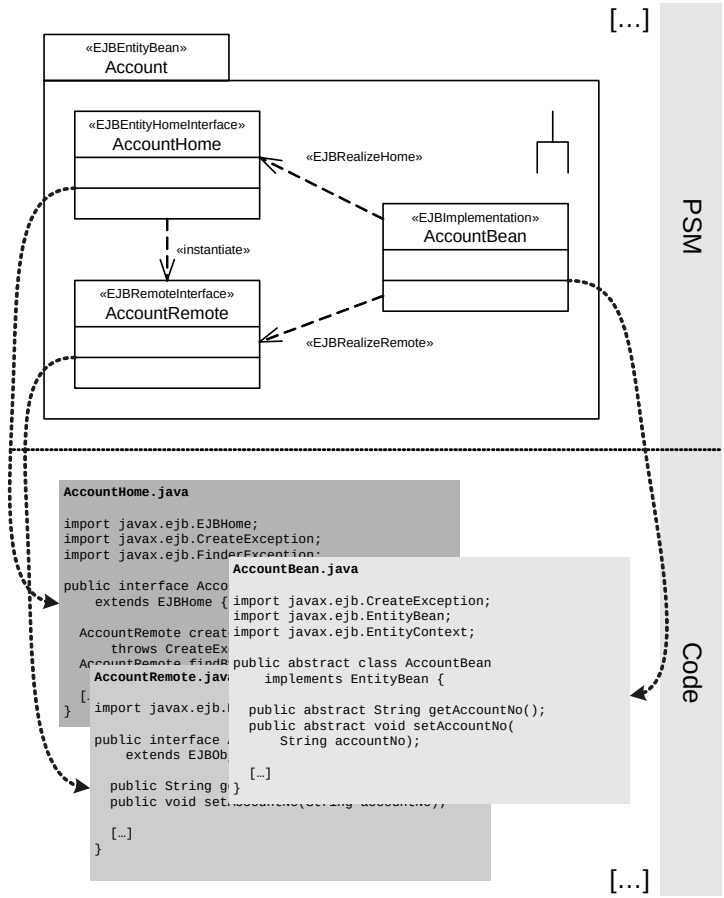


Abb. 2.9 Transformationsmuster konkret – vom PIM zum PSM

Mit dem *UML Profile for EDOC* können wir von einer konkreten technologischen Plattform für verteilte Objekttechnologie abstrahieren. Es wäre also auch möglich gewesen, das so notierte PIM in ein PSM für das CORBA Components Model (CCM) [CCM] zu transformieren.

Betrachten wir nun das Ergebnis einer derartigen Transformation für ein einzelnes Modellelement, wie in Abb. 2.9 illustriert. Man sieht an diesem Beispiel recht deutlich, dass das PIM auf einer höheren Abstraktionsebene – quasi ein höhere semantische Ebene – angesiedelt ist: Die „Kennzeichnung“ «Entity» der Klasse Account trägt die Semantik, die zur Abbildung dieser Klasse in eine EntityBean des PSM führt.

Abb. 2.10 Transformationsmuster konkret – vom PSM zum Code



Sie wird expandiert in die BeanClass (Implementierungsklasse) sowie RemoteInterface (Komponentenschnittstelle) und HomeInterface (Fabrikschnittstelle), so wie es das Komponentenmodell bzw. die EJB-Spezifikation verlangt. In der Abbildung nicht dargestellt sind entsprechende Operationen zum EJB-

Lebenszyklus (`ejbCreate`, `ejbActivate`, …), diese werden ebenfalls erzeugt.

Werfen wir zuletzt noch einen Blick auf die Artefakte, die generiert würden. In Abb. 2.10 ist der Java-Quellcode der bereits angesprochenen Klassen bzw. Interfaces skizziert.

Neben den in der Abbildung dargestellten Klassen bzw. Schnittstellen würden zusätzlich Deployment-Deskriptoren (`ejb-jar.xml`, `application.xml`, sowie herstellerspezifische Deskriptoren), SQL-Skripte (etwa Skripte zum Aufsetzen des Datenbankschemas und zum Einspielen des initialen Datenbestandes), Testskripte und ähnliches mehr generiert werden. Der Kreativität sind hier zunächst einmal keinerlei Grenzen gesetzt, der Übersichtlichkeit wegen haben wir uns jedoch an dieser Stelle auf die Quellcode-Anteile beschränkt.

2.2.4 Standards im Dunstkreis der MDA

An dieser Stelle wird ein Überblick über Standards gegeben, die eine wichtige Rolle im Zusammenhang mit MDA spielen. Auf einige werden wir im Verlauf des Buches noch näher eingehen, andere werden nur kurz vorgestellt und dann auf die entsprechende Literatur verwiesen.

MOF

Die Meta Object Facility (MOF) der OMG ist eine modellbasierte Sprache zur Definition von Metamodellen. So sind beispielsweise sämtliche Standards des UML-2.x-Stacks mit ihr beschrieben. Kapitel 3 (Modellierung) geht näher auf dieses mächtige Werkzeug dem „Metamodell für Metamodelle“ ein.

UML

UML, die Unified Modeling Language der OMG, ist das Mittel der Wahl zur Erstellung der Modelle innerhalb der MDA. Kapitel 3 geht näher auf die Entwicklung der Sprache ein, Anhang A liefert eine Referenz und schnellen Überblick über alle Diagrammarten.

XMI

Das XML Metadata Interchange (XMI) definiert eine Abbildung der MOF auf XML. Damit ermöglicht es den standardisierten Austausch von beliebigen (Meta-)Modellen zwischen Tools wie Transformatoren, Modellierungswerkzeugen, Codegeneratoren usw. Der standardisierte Austausch von Modellinformationen ist eine der Grundvor-

aussetzungen zum Aufbau einer funktionierenden MDA-Infrastruktur. Kapitel 3 geht näher auf diesen Teil der UML-Spezifikation ein.

OCL

Eine Möglichkeit der deklarativen Präzisierung von Modellen bietet die Object Constraint Language (OCL). Sie macht es möglich, Bedingungen an die Verwendung von Modellelementen zu stellen, die zur Laufzeit des Modells zu jedem Zeitpunkt erfüllt werden müssen. Neben den UML-Action-Semantics ist sie damit eines der UML-eigenen Mittel zur Präzisierung dynamischer Aspekte in UML-Modellen. Auf OCL und Action Semantics wird in Kapitel 3 – Modellierung im Detail eingegangen.

QVT

Queries Views and Transformations (kurz QVT) ist der kommende Standard zur Beschreibung von Modelltransformation auf Basis der MOF und OCL.

HUTN

Die Human-Usable Textual Notation (HUTN) liefert das Mapping von UML Modellen auf eine textuelle Notation und ist damit prinzipiell vergleichbar mit dem bereits erwähnten XMI. Im Vergleich mit XMI setzt sie jedoch den Schwerpunkt auf die Lesbarkeit und besserem Verständnis des Textes durch Menschen. Die zugehörige Spezifikation, die momentan in Version 1.0 vorliegt, findet sich unter [HUTN].

2.3 Ideen, Anleihen und verwandte Ansätze

Bei der Formulierung der MDA hat die OMG nicht auf der grünen Wiese angefangen, sondern vielmehr verschiedene Ideen direkt oder indirekt aufgegriffen, zu einem geschlossenen Konzept vereint und daraus einen Standard gemacht. Die in Abb. 2.11 dargestellten Ideen, Anleihen und verwandten Ansätze gehören dazu, wir wollen sie an dieser Stelle erläutern und in den MDA-Kontext einordnen.

Der Compiler als Vorbild?

Einschieben wollen wir aber zunächst die vielleicht offensichtlichste Anleihe: den *Compiler (dt. Übersetzer)*. Die Arbeitsweise eines Compilers (siehe Skizzierung in Abb. 2.12) ähnelt dem Transformationsansatz der MDA: Am Anfang steht ein Quellprogramm, welches – via diverser Analyse- und Generierungsschritte – in ein

Zielprogramm überführt wird. Wie wir im Verlaufe dieses Buches noch sehen werden, gleichen die erwähnten Zwischenschritte in signifikanter Weise denen eines Transformators bzw. Generators in der modellgetriebenen Welt. So arbeiten beispielsweise beide Verfahren i. d. R. auf einer *Zwischenrepräsentation (engl. intermediate representation)*.

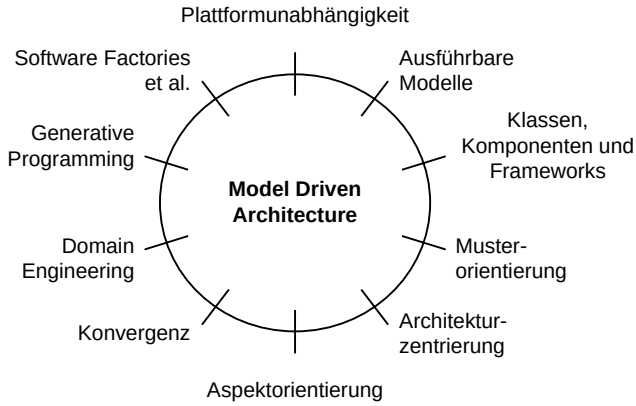


Abb. 2.11 Ideen, Anleihen und verwandte Ansätze

Aufgrund dieser Gleichartigkeit spricht man in diesem Zusammenhang daher auch von einem *Model-Compiler*.

Model-Compiler

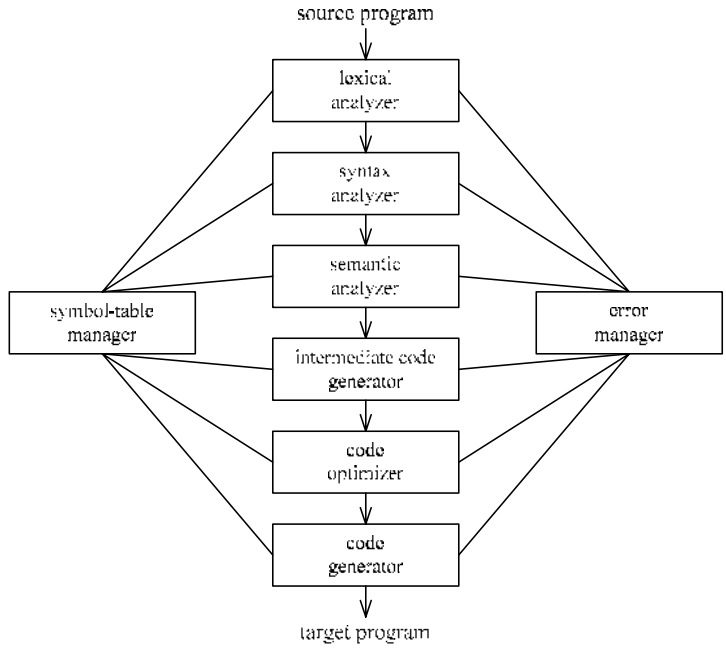
Aber wo liegen dann die Unterschiede, die eine neue Herangehensweise rechfertigen? Nun, um es etwas flapsig zu formulieren: In modellgetriebenen Ansätzen wie der MDA wird man selbst zum *Compilerbauer*, indem man die notwendigen „Übersetzungsregeln" selbst definiert.

Macht sich also womöglich der Besuch der Vorlesungen zu *Theoretischer Informatik* und *Übersetzerbau* am Ende doch noch bezahlt? ;-)

Die Antwort an dieser Stelle ist ein entschiedenes *Jein!*. Die Kenntnis über die theoretischen Hintergründe der verwendeten Techniken ist hilfreich zur Vertiefung des Gesamtverständnisses aber keine Einstiegsvoraussetzung. Ein wesentliches Ziel der MDA ist es schließlich, die notwendigen Transformationsregeln möglichst einfach formulieren zu können. In der MDA-Vision gehört das Arbeiten mit Tranformationen demnach auch zum alltäglichen Handwerkszeug eines Entwicklers. Die damit einhergehenden Verschiebungen im Qualifikationsprofil eines Entwicklers wollen

wir hier allerdings nicht weiter vertiefen und verweisen stattdessen auf die nachfolgenden Kapitel.

Abb. 2.12 Die Phasen eines Compilers (nach [ASU86])



2.3.1 Plattformunabhängigkeit

Die Idee der *Plattformunabhängigkeit* ist in der Informatik nicht neu und eng mit dem Wunsch nach Portierbarkeit von Anwendungen verbunden. So erreichen beispielsweise die höheren Programmiersprachen die Unabhängigkeit von der „Plattform“ Prozessor, indem es der Compiler ermöglichte, die Programme in mehr als eine prozessor-spezifische Maschinensprache zu übersetzen.

Die Portierbarkeit von Anwendungen ist allerdings nicht nur vom Prozessor, sondern von anderen Komponenten der Ausführungsumgebung abhängig, etwa weiteren Hardwarebestandteilen oder aber dem Betriebssystem. Das Konzept der *Virtuellen Maschine*, so wie es beispielsweise bei der Programmiersprache Java umgesetzt wird, kann hier für die gewünschte Plattformunabhängigkeit sorgen. Eine Virtuelle Maschine kapselt den Zugriff auf die Ausführungsumbebung durch die Bereitstellung einer spezifizierten Schnittstelle. Jede Nutzung dieser Schnittstelle interpretiert die Virtuelle Maschine und übersetzt sie in plattform-spezifische Instruktionen.

Beim MDA-Ansatz wird diese Idee durch die plattformunabhängigen Modelle (PIMs) realisiert. Die Wahl der Bezugsplattform ist dabei zunächst einmal nicht eingeschränkt. Ein Beispiel für zwei unterschiedliche Plattformen, die allerdings in fast jeder Beschreibung der MDA (also auch hier) herhalten müssen, sind die Plattformen Java EE und .NET.

Aufgrund der in Abschnitt 2.1.3 geschilderten Integrationsproblematik liegt es jedoch näher, generell Middleware-Technologien unter die zu abstrahierenden Plattformen zu fassen. Kapitel 4 widmet sich dem Gedanken der Plattform und der resultierenden Abhängigkeiten näher.

2.3.2 Ausführbare Modelle

Insbesondere im Bereich der Embedded Systems (dt. etwa: Eingebettete Systeme) wird die Idee der ausführbaren Modelle bereits erfolgreich umgesetzt. Dafür wurden eigene Modellierungssprachen entwickelt, die teilweise mit Bestandteilen der Unified Modeling Language (UML) kombiniert wurden. Hier sind beispielsweise die Real-Time Object-Oriented Modeling Language (ROOM Modeling Language) [SGW94] sowie die Specification and Description Language (SDL) [SDL] zu nennen. Diese Sprachen sind so ausdrucksstark und präzise, dass die damit beschriebenen Modelle direkt von einem „Model-Compiler“ in ausführbare Programme übersetzt werden können.

Executable UML

Ein entsprechender Ansatz, der für sich in Anspruch nimmt, nicht nur im Embedded Bereich zu funktionieren, ist der der *Executable UML (xUML)* von Mellor und Balcer [MB02]. Der Ansatz beschreibt, wie man die UML so präzise verwenden kann, dass die damit erstellten Modelle ausführbar werden. Ähnlich wie beim MDA-Ansatz gibt es bei diesem Ansatz ein plattform-unabhängiges Modell. Dieses wird allerdings unmittelbar in eine ausführbare Form überführt, der Zwischenschritt des plattform-spezifischen Modells wird also übersprungen.

Vergleich mit MDA

In der MDA besteht zwar im Grunde dasselbe Ziel, nämlich ein System so präzise und vollständig durch Modelle zu beschreiben, dass eine lauffähige Anwendung aus ihnen generiert werden kann, die Umsetzung dieses Gedankens ist jedoch von der der xUML verschieden. Während xUML anstrebt, *ein* turing-vollständiges UML Profil zu schaffen, mit dem alle Systeme lauffähig beschrieben werden können, verfolgt die MDA den Grundsatz die Informationen, die zur Generierung der letztendlichen Anwendung benötigt werden,

aus *vielen* Modellen herauszuziehen, die zur Beherrschung der unterliegenden Komplexität nur jeweils bestimmte Aspekte des Systems darstellen (vergleiche dazu das Konzept der Domänen-spezifischen Sprachen, DSL). Die Extraktion der notwendigen Informationen aus den Einzelmodellen und die daraus resultierende Transformation von Modellen in immer technik-nähere Darstellungen bis hin zu auf einer Plattform lauffähigen Repräsentation (z. B. Quellcode) geschieht dabei idealerweise vollautomatisiert.

2.3.3 Klassen, Komponenten und Frameworks

Keine Angst, die grundsätzlichen Ideen hinter Klassen, Komponenten und Frameworks wollen wir an dieser Stelle nicht langwierig erörtern, hier sei auf die einschlägige Literatur dazu verwiesen (etwa [Boo93] und [HS99]). Sie hier dennoch aufzuführen erfolgt aus dem Umstand, dass die Ideen hinter Objektorientierung (OO), Komponentebasierter Softwareentwicklung (Component-Based Development, CBD) und Frameworks mit der MDA nicht obsolet werden. Im Gegenteil: sie erleben in gewisser Weise eine Rekonvaleszenz.

Fragt sich, wovon sich diese Ansätze genesen. Nun, neben dem Fortschritt den diese Ansätze uns bezüglich der Komplexitätsbewältigung gebracht haben (vgl. Abschnitt 2.1.1) stehen sie in der Kritik ihre ursprünglichen Versprechen nicht eingelöst und damit die an sie gestellten Erwartungen nicht erfüllt zu haben. Exemplarisch seien hier zwei der wesentlichen Kritikpunkte herausgegriffen:

- **Mangelnde Wiederverwendbarkeit:** OO, CBD und Frameworks haben uns bezüglich der Strukturierung von Software-Systemen vorangebracht, aber kaum Erfolge für die Wiederverwendbarkeit von Software erzielt. Als Grund dafür ist ihre derzeitige mangelnde Anpassbarkeit zu nennen, die ihre Einsatzmöglichkeiten stark einschränkt, denn selten lässt sich beispielsweise eine Komponente ohne Anpassungen in verschiedenen Kontexten zum Einsatz bringen.

  Wie wir eingangs dieses Kapitels in unserem Problemdiskurs bereits festgestellt hatten (siehe Abschnitt 2.1.3), ist Software hinsichtlich fachlicher Belange und erst recht in Bezug auf ihre Ausführungs-Technologie einem schnell voranschreitenden Alterungsprozess unterworfen. Offensichtlich gilt diese Aussage auch implizit für die Einheiten aus der eine Software besteht. Betrachten wir uns etwa den Komponentenansatz, so müssen wir eingestehen, dass die Legostein-Metapher offensichtlich falsch ist: Während die „Schnittstelle“ der *echten* Legosteine seit 1963 stabil ist und sie sich in nahezu beliebigem Kontext

einsetzen lassen, müssen die Schnittstellen ihrer softwaretechnischen Pendants aufgrund von divergierenden technischen oder fachlichen Kontextanforderungen und/oder Änderungszyklen dazu stetig angepasst werden.

Wiederverwendbarkeit durch Variabilität in der MDA

Mit einem generativen Ansatz wie bei der MDA lässt sich die notwendige Variabilität erreichen: Auf Ebene des PIM werden Klassen, Komponenten und Frameworks plattform-unabhängig und funktional variabel modelliert. Erst durch den Transformationsschritt werden Plattform und die konkrete Funktionalität festgelegt. Wie wir weiter unten unter den Titeln „Domain Engineering“ und „Generative Programming“ sehen werden, lässt sich diese Herangehensweise verallgemeinern und diese *Wiederverwendbarkeit durch Variabilität* auf eine breitere Basis stellen.

- **Unzureichende Notationsmöglichkeiten für Komponentenstrukturen:** Exemplarisch sei hier das Komponentenmodell EJB angeführt. Es fußt auf der Programmiersprache Java, die selbst über keine geeignete Notation wesentlicher Konzepte von Komponenten wie *Ports*, *Konnektoren*, *benötigte* und *bereitgestellte Schnittstellen* usw. verfügt. Stattdessen ist man dazu gezwungen schwer lesbare, redundanz-behaftete Deployment-Deskriptoren zu erstellen (vgl. Abschnitt 2.1.3 – Descriptor Hell).

  Mit der MDA bieten sich hier zwei Alternativen: Entweder man erstellt ein eigenes Metamodell, um diese Elemente zu notieren oder man setzt auf die neuen Spezifizierungsmöglichkeiten der UML 2.0. Im Sinne einer präskriptiven Verwendung lassen sich die damit definierten Modelle zur Erzeugung des benötigten Gluecodes (Deskriptoren, …) heranziehen.

2.3.4 Musterorientierung

Bevor der Begriff des Musters (engl. Pattern) in der Software-Welt populär wurde, wurde er bereits im Kontext von Gebäudearchitekturen durch Arbeiten von Christopher Alexander Ende der 70er-Jahre geprägt. In [AIS77] schrieb er dazu:

> *„Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.“*

Die Überführung dieser Idee in das Software-Engineering geht auf Beck und Cunningham [BC87] sowie die Gang of Four [GOF] zurück, die Muster zunächst als Grundlage für den Software-Entwurf ins Spiel brachten. Mittlerweile ist das Konzept des Musters auf viele weitere Bereiche des Software-Entwicklungsprozesses ausgeweitet worden, nachfolgend daher eine Auflistung der gängigsten Varianten:

- *Architekturmuster* – je nach Quelle auch synonym für Architekturstil verwendet – sind Muster auf oberster Entwurfsebene. Sie beschreiben grundsätzliche Organisationsprinzipien für die Architektur einer Software.
  →Beispiele: Pipes & Filters, Blackboard, Model-View-Controller (MVC)
- Die bereits angesprochenen *Entwurfsmuster* sind auf einer niedrigeren Abstraktionsstufe als Architekturmuster angesiedelt. Sie werden bei der Verfeinerung von Subsystem herangezogen.
  →Beispiele: Observer, Factory, Composite
- Ein *Idiom* beschreibt eine Lösung eines speziellen Implementierungsproblems in einer dedizierten Programmiersprache.
  →Beispiele: Typesafe Enumeration sowie Multiple Inheritance (Java), Reference Counter (C++)
- Die in der Fachdomäne beheimateten *Analysemuster* [Fow96] liefern erprobte Analyseergebnisse in Form von Objektmodellen (sic!).
  →Beispiele: Party, Accountability, Organization Hierarchies
- *Archetypmuster* [AN04] führen die Idee der Analysemuster weiter: Sie beschreiben grundsätzliche, ursprüngliche Konzepte für die Domäne der Unternehmensanwendungen. Das sind Dinge oder Konstellationen, die immer wieder in dieser Domäne anzutreffen sind, etwa das Prinzip der Doppelten Buchführung oder der Zusammenhang von Produkt, Kunde und Bestellung. Der Katalog dieser Muster ist bereits für die Verwendung in einem modellgetriebenen Vorgehen wie der MDA vorbereitet, d. h., die entsprechenden Referenzmodelle können auf die spezifischen Anforderungen eines bestimmten Anwendungsexemplars angepasst werden – sie sind damit im Sinne eines domänenorientierten Ansatzes (siehe Abschnitt 2.3.8) variabel gehalten.

Mit den letztgenannten Archetypmustern wurde bereits grob skizziert, welche Rolle Muster in der MDA einnehmen. Muster werden

entweder manuell vom Modellierer deklariert oder aber vollautomatisch erkannt und dann jeweils vom Transformator expandiert, d. h. in eine Implementierung überführt.

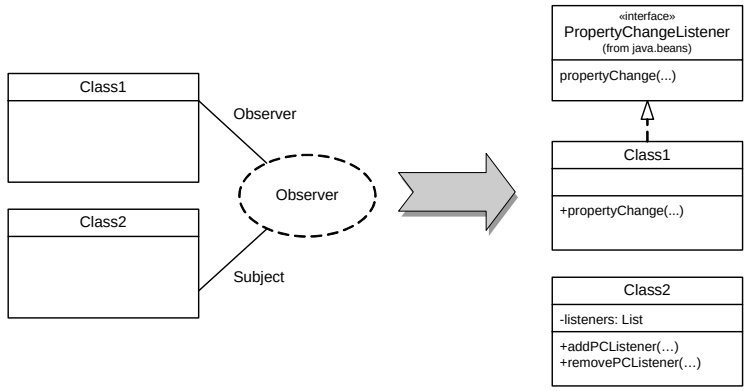


Abb. 2.13 Replizierung eines Musters

In diesem Zusammenhang spricht man auch von *Pattern Replication (dt. Muster-Replizierung)*, da ein Muster nur einmal erfasst und seine Umsetzung dann beliebig oft kopiert wird. Abb. 2.13 veranschaulicht dies beispielhaft für das Observer-Pattern, welches in eine Java-Implementierung überführt wird.

2.3.5 Architekturzentrierung

Zwar existiert nicht *das* architekturzentrierte Paradigma, allerdings werden Software-Entwicklungsprozesse entsprechend bezeichnet, wenn sie gewisse Kriterien bezüglich ihrer Herangehensweise an die Entwicklung von Software erfüllen. In diesen Prozessen oder Vorgehensweisen wird die Planung und Konzeption der Architektur als kritischer Entwicklungsschritt betrachtet. Daher stellt man frühzeitig ein Architekturmodell auf und optimiert es fortlaufend. Wichtige Bestandteile eines solchen Modells sind u. a. Konzepte, Abstraktionen, Schnittstellen und Muster, an denen sich die Anwendungsentwickler orientieren können, die also eine verbindliche Referenz über den gesamten Entwicklungsprozess darstellen. Ein Vertreter dieser Prozessart ist der Unified Software Development Process [JBR98], den wir an dieser Stelle als exemplarisches Beispiel nennen wollen.

Oftmals ergänzt wird das Architekturmodell durch eine Referenzimplementierung bzw. einen Architekturprototypen. Durch eine Art schematische, schablonenhafte Entwicklung der Anwendung als Show-Case möchte man so die Konformität zur gewünschten Architektur sichern. Welche Vorgaben dabei an eine valide Architektur gestellt werden manifestiert sich oft in Vorgaben zu den anzuwen-

denden Komponentenschnitten und Designmustern sowie in Angaben, die etwa die Interaktionsschnittstellen zwischen den einzelnen Schichten vorschreiben.

MDA begegnet diesem Anliegen durch die Angabe eines fachlichen Architekturmetamodells auf der fachlichen und plattform-unabhängigen sowie der Vorgabe von Designprinzipien und technischen Idiomen durch Platform Description Models und Transformationsvorschriften auf der plattform-spezifischen Seite der Modellkette (diese Konzepte werden in Kapitel 4 noch ausführlich dargestellt). Die tatsächliche Durchsetzung einer einheitlichen Architektur wird in der MDA durch die weitestgehende Automatisierung der Transformationen zwischen plattform-unabhängigen zu plattform-spezifischen Modellen erreicht.

2.3.6 Aspektorientierung

Häufig sind bestimmte Belange eines Systems über den Programm-Code einer Anwendung verstreut und lassen sich, obwohl in gewisser Weise zusammengehörig, nur unzureichend bis gar nicht in ein einzelnes Modul pressen. Man nennt solche Belange *Crosscutting Concerns (dt. Querschnittsbelange)*. Meist sind solche Crosscutting Concerns nicht funktional sondern eher technisch geprägt. Typische Kandidaten dafür sind beispielsweise Logging, Caching, Synchronisation und Pooling.

Aspekt-orientierte Programmierung

Bei der *Aspektorientierten Programmierung (Aspect Oriented Programming, AOP)* [KLM+97] werden diese Crosscutting Concerns in Module – in der AOP-Welt *Aspects (dt. Aspekte)* genannt – gekapselt. Zur Implementierung dieser Aspekte stehen dem Entwickler erweiterte bzw. neue Programmiersprachkonstrukte zur Verfügung. Mit ihnen spezifiziert er die auszuführende Funktionalität selbst und die *Join Points (dt. Verbindungspunkte)*, an denen sie ausgeführt werden soll. Ein *Aspect Weaver (dt. Aspektweber)* fügt dann die Aspekte und den (fachlichen) Code wieder zusammen (vgl. Abb. 2.14).

Ein Applikationsserver – oder allgemeiner gesprochen eine Middleware – weist ebenfalls eine Aspektorientierung auf. Hier werden technische Belange (Persistenz, Transaktionalität, …) von der Laufzeitumgebung über entsprechende Schnittstellen zur Verfügung gestellt. Im Falle der Enterprise JavaBeans (EJB) kann dies sogar rein deklarativ über so genannte Deployment-Deskriptoren geschehen.

Die MDA setzt diese Idee um, indem sie zum einen das von technischen Belangen freie PIM und das um selbige angereicherte PSM trennt. Modellelemente des PIMs werden mit den Aspekten „markiert“, die auf sie angewendet werden sollen. Im Falle der Persistenz könnte dies zum Beispiel die Verwendung des Stereotypen `«Entity»` sein. Zum anderen geschieht die Beschreibung der technischen Umsetzung des Aspektes in eine lauffähige Anwendung zentral durch die Definition von Abbildungsregeln, die beschreiben, wie der Aspekt technisch auf einer bestimmten Plattform zu realisieren ist (in unserem Beispiel etwa die Verwendung von EJBs).

Umsetzung des Gedankens in der MDA

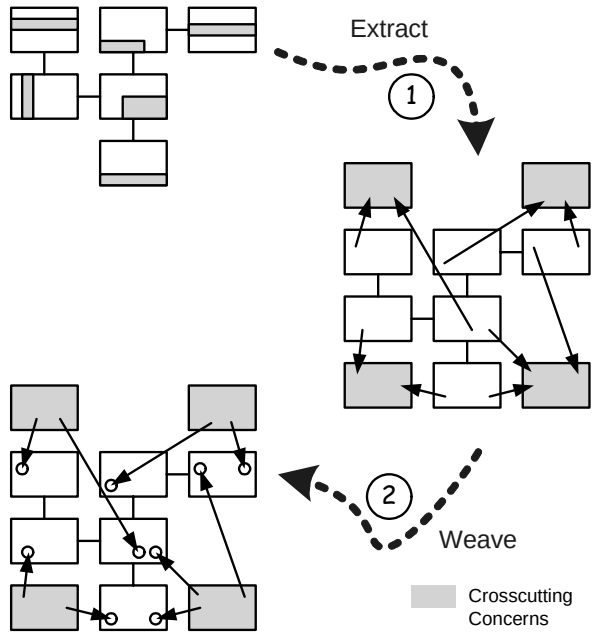


Abb. 2.14 Aspektorientierung

2.3.7 Konvergenz

Unter dem Begriff der Konvergenz wird innerhalb des Software-Engineering die ganzheitliche Betrachtung von Geschäfts- und Software-Systemen verstanden. Der bekannteste Vertreter für diesen Ansatz ist das von Taylor propagierte *Convergent Engineering (CE)* [Tay95]. Im CE wird die Ansicht vertreten, dass der Aufbau eines Geschäftsprozesses und das Design der ihn unterstützenden Anwendungen sich nicht wesentlich voneinander unterscheiden sollten.

Convergent Engineering

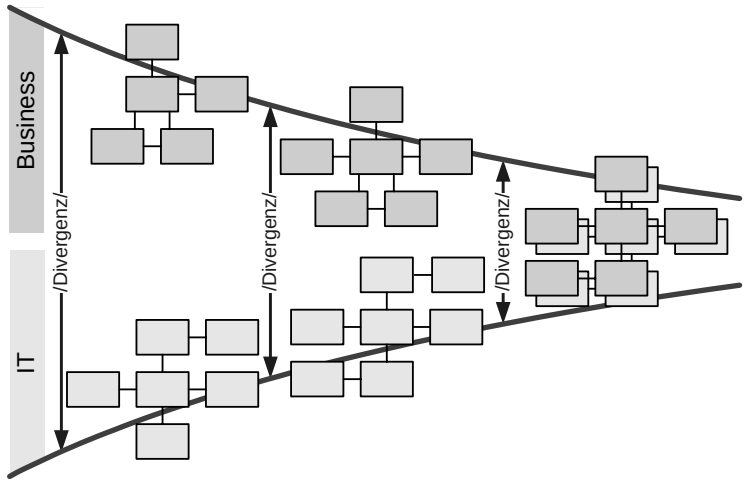
Dementsprechend wird angestrebt Geschäftssystem und IT-System konvergieren zu lassen (siehe Abb. 2.15).

Dazu wird die Invariante aufgestellt, dass zu jedem Schlüsselobjekt eines Prozesses genau eine korrespondierende Software-Komponente existiert. Im Wesentlichen existieren drei Kategorien solcher Komponenten:

- Organisationen,
- Prozesse
- und Ressourcen.

Zusätzlich werden Regeln definiert, die bestimmen, wie diese Komponenten kombiniert werden dürfen sowie das Zusammenspiel unter ihnen festlegen.

Abb. 2.15 Prinzip des Convergent Engineering



Convergent Architecture

Die *Convergent Architecture (CA)* von Hubert [Hub01] ist eine Implementierung dieses Ansatzes. Sie stützt sich dabei auf so genannte *Konvergente Komponenten (engl. Convergent Components)*, die dazu verwendet werden ein Unternehmen softwaretechnisch abzubilden. Diese Komponenten sind unabhängig von jeglicher Technologie und werden mittels *Technology-Projection* auf Komponentenmodelle wie etwa Java EE abgebildet. Die Convergent Architecture beschreibt darüber hinaus weitere Konzepte, Techniken, Vorgehen sowie konkrete Werkzeuge zur Umsetzung des Ansatzes, auf die wir an dieser Stelle nicht weiter eingehen wollen.

Der interessierte Leser sei hier auf die angegebene Literatur verwiesen.

Konvergenz in der MDA

Die beschriebene Konvergenz ist kein primäres Ziel der MDA. Allerdings wird durch die höhere Abstraktionsebene, auf der die PIMs angesiedelt sind, die sprachliche Kluft zu den Geschäftsmodellen bereits wesentlich verkleinert. Darüber hinaus sichert die MDA durch die geforderte Rückverfolgbarkeit von Transformationen explizit zu, dass sich die im CIM modellierten Anforderungen, die von sich aus eine hohe Affinität zu den unterstützten Geschäftsprozessen aufweisen, jederzeit in den Konstrukten des PIMs bzw. PSMs wiederfinden lassen können müssen und umgekehrt.

Geeignet angewendet kann die MDA als ein „Werkzeug" für das Erreichen von Konvergenz verstanden werden. Im Verlaufe dieses Buches werden wir noch entsprechende Beispiele dafür sehen, wie die konkrete Umsetzung der Idee verwirklicht werden kann.

2.3.8 Domain Engineering

Beim *Domain Engineering* [DE] versucht man die Wiederverwendung von Entwicklungswissen einer spezifischen Domäne auf eine systematische Basis zu stellen. Dazu werden die Erfahrungen, die man bei der Erstellung der Systeme dieser Domäne erhalten hat, gesammelt, organisiert und festgehalten. So erhält man einen *Wiederverwendungsbestand (engl. Reusable Assets)*, auf den bei der Entwicklung eines neuen Systems der gleichen Domäne zurückgegriffen werden kann. Während man sich bei der konventionellen Software-Entwicklung genau einem System widmet, liegt der Fokus beim Domain Engineering also auf einer ganzen *Systemfamilie (engl. System Family).*

Das Domain Engineering unterteilt sich in:

- *Domain Analysis (dt. Domänen-Analyse)*
- *Domain Design (dt. Domänen-Entwurf)*
- *Domain Implementation (dt. Domänen-Implementierung)*

Domänen-Analyse

Die *Domänen-Analyse* soll dem besseren Verständnis (der Durchdringung) einer Domäne dienen. Dazu wird zunächst eine Domäne ausgewählt und abgegrenzt. Dann werden die Gemeinsamkeiten und Variabilitäten existenter Systeme der Domäne herausgearbeitet. Das Resultat dieser Analyse-Aktivität ist ein kohärentes *Domänenmodell (engl. Domain Model)*, in der die gemeinsamen und variablen

Eigenschaften der Systeme der betrachteten Domäne festgehalten sind. Das Modell beschreibt zudem die Bedeutung und die Abhängigkeiten zwischen diesen Eigenschaften.

Domänen-Entwurf

Während des *Domänen-Entwurfs* wird mit den Erkenntnissen aus der Domain Analysis eine gemeinsame Architektur – eine *Referenzarchitektur (engl. Common/Reference Architecture)* – für die Domäne entworfen. Mit ihr erhält man einen adaptierbaren Entwurf und damit die Grundlage für den ebenfalls in dieser Phase zu erstellenden Produktionsplan. Dieser Plan beschreibt, wie ein konkretes System der fokussierten Domäne aus dem Wiederverwendungsbestand abgeleitet werden kann.

Domänen-Implementierung

Bei der *Domänen-Implementierung* wird schließlich die angesprochene Architektur samt ihrer Komponenten so umgesetzt, dass sie generisch bzw. generativ bezüglich der Gemeinsamkeiten und Variabilitäten wiederverwendet werden kann. Dazu passend wird eine *Domänen-Spezifische Sprache (engl. Domain-Specific Language, DSL)* entworfen. Diese DSL ist so aufzubauen, dass sie in kompakter Form mit Begriffen und Konzepten der Domäne die Spezifikation jedes zulässigen Systems der Domäne ermöglicht.

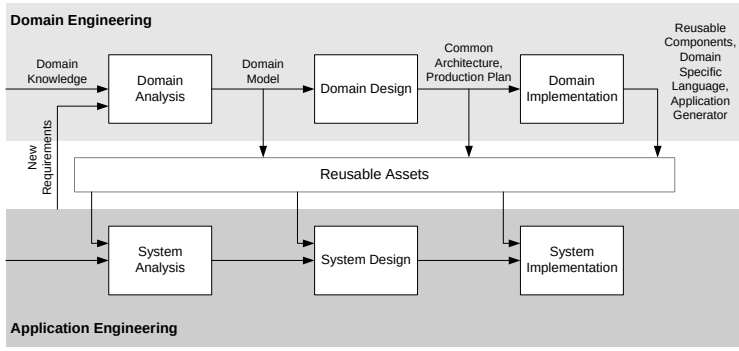


Abb. 2.16 Zusammenhang von Domain und Application Engineering

Application Engineering

Im *Application Engineering* wird der so geschaffene Wiederverwendungsbestand zur Entwicklung eines neuen Systems genutzt. Die im Application Engineering neu gewonnenen Anforderungen werden dem Domain Engineering zurückgemeldet. Zusammengefasst spricht man beim Domain Engineering deshalb auch von der Entwicklung *für* und beim Application Engineering von der Entwicklung *mit* Wiederverwendung. Dieser Zusammenhang ist in vereinfachter Form in Abb. 2.16 dargestellt.

Die MDA verfolgt ganz ähnliche Wiederverwendungsansätze. So sieht sie es beispielsweise vor Abbildungsregeln wiederverwendbar zu machen. In diesen Abbildungsregeln kann eine gemeinsame Architektur der Mitglieder einer Systemfamilie festgehalten werden. Die domänen-spezifischen Sprachen können beispielsweise über UML-Profile umgesetzt werden. Zurzeit adressiert die MDA allerdings eher die technische Variabilität von Systemfamilien, das Transformationsmuster ist aber prinzipiell auch auf funktionale Variabilität übertragbar. Domain Engineering und MDA können sich jedenfalls sinnvoll ergänzen, wie wir im Kapitel über die Einbettung von MDA in bestehende Software-Prozesse (Kapitel 6) sehen werden, wo uns die eben beschriebenen Konzepte unter dem Begriff der *Software-Produktlinien (engl. Software Product-Lines, SPL)* (siehe auch [BKPS04]) wieder begegnen werden.

2.3.9 Generative Programming

Generative Programming (GP) stützt sich auf das eben erläuterte Domain Engineering. Es geht auf eine Arbeit von Czarnecki und Eisenecker zurück [CE00] und beschreibt Vorgehen und Techniken, sodass die Entwicklung eines Mitglieds einer Systemfamilie (siehe oben) vollautomatisch aus Anforderungsspezifikation, Konfigurationswissen und vorgefertigten, anpassbaren Komponenten erfolgen kann. Das in Abb. 2.17 dargestellte Generative Domänenmodell illustriert den grundsätzlichen Ansatz.

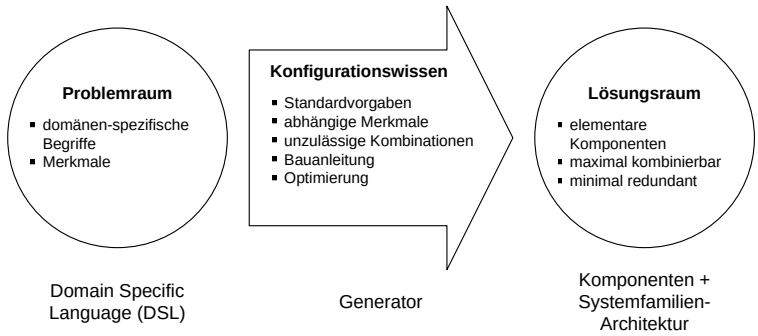


Abb. 2.17 Generatives Domänenmodell

Der *Problemraum* enthält domänen-spezifische Begriffe und Merkmale, die in der DSL zusammengefasst sind. Mit dieser DSL wird ein Systemfamilienmitglied spezifiziert. Das *Konfigurationswissen* wird durch einen Generator umgesetzt. Dieser ergänzt zunächst die Spezifikation durch Standardvorgaben und prüft Abhängigkeiten und unzulässige Kombinationen ab. Dann setzt er die Spezifikation

um, indem er vorgefertigte Komponenten des *Lösungsraums* zu einem ausführbaren System zusammensetzt.

Dadurch, dass das Generative Programming auf dem Domain Engineering fußt, sind die oben genannten Anmerkungen auch hier gültig. Eine zusätzliche Gemeinsamkeit ist in dem angestrebten Automatisierungsgrad auszumachen. Bei beiden Ansätzen soll eine fachlich geprägte Systemspezifikation zur vollständigen Überführung in eine ausführbare Anwendung dienen.

2.3.10 Software-Factories et al.

Von den vorgestellten Ansätzen ist der der *Software-Factories (dt. Software-Fabriken)* sicherlich zuvorderst als Konkurrenz zu MDA zu sehen. Konzipiert wurde er von Jack Greenfield, Keith Short, Steve Cook und Stuart Kent. Niedergeschrieben haben sie ihn im gleichnamigen Buch [GSCK04]. Bevor sie 2002 zu Microsoft wechselten, waren die Autoren allesamt an zahlreichen OMG-Spezifikationen beteiligt, insbesondere an Einreichungen zur UML 2, was der genannten Lektüre unschwer zu entnehmen ist. Ihre Vision von moderner Software-Entwicklung wollen sie nunmehr im Produkt *Microsoft Visual Studio Team System* Realität werden lassen.

Wie die MDA bedient sich der Ansatz der Software-Factories ausgiebig an den Konzepten von modellgetriebener Entwicklung, von Produktlinienansätzen sowie Musterorientierung und Frameworks. Mit der Metapher der *Fabrik* unterstreichen die Autoren die Rolle der für den Ansatz notwendigen Infrastruktur – also quasi die Fertigungsstraße, um im Jargon industrieller Fertigung zu bleiben. Die Tatsache, dass Microsoft hinter diesem Ansatz steht, spricht zusätzlich dafür, dass die Fokussierung auf einer geschlossen, einfach zu bedienenden Werkzeugkette liegen wird. Geht man nach den Ausführungen der Autoren, so wird ein Hauptaugenmerk auf die Möglichkeit zur Erstellung eigener domänen-spezifischer Sprachen gelegt werden. Die UML bzw. die MOF als Basis dafür heranzuziehen lehnen die Autoren ab: UML habe sich zwar zu Dokumentationszwecken bewährt, als formale Grundlage für domänenspezifische Sprache sei sie aber untauglich.

Auch wenn die jüngere Vergangenheit gezeigt hat, dass Microsoft dazu fähig ist, offene Standards zu etablieren bzw. sie zu unterstützen (siehe etwa SOAP, BPEL, o. Ä.), ist zu befürchten, dass wesentliche Teile des Ansatzes nicht offen gelegt werden und für potenzielle Kunden damit die Gefahr eines Vendor-Lock-Ins besteht.

Im abschließenden Kapitel dieses Buches – wenn es um die Bewertung der MDA und ihre zukünftigen Herausforderungen geht – und in *Kapitel 10 - /lost+found* werden wir uns diesem Themenkomplex und dem Vergleich der beiden Ansätze noch einmal detaillierter widmen.

Et alii?

Neben der gerade diskutierten Software-Factories verfolgen eine ganze Reihe weiterer Ansätze die Ideen modellgetriebener Entwicklung, domänen-spezifischer Sprachen und Produktlinien. Stellvertretend seien hier Language-Driven Development (LDD), Language-Oriented Programming (LOP) und Intentional Programming (IP) genannt. Eine weitere Diskussion dieser Ansätze würde allerdings den Umfang dieses Buches sprengen, wir verweisen daher auf entsprechende Online-Quellen, wie etwa das Code Generation Network [CGN].

2.4 Pragmatische Sichten auf MDA

Nach diesem kurzen Überblick über die Konzepte der MDA werden wir uns nun der Praxis zuwenden. Dabei soll vor allem untersucht werden, warum sich eine Beschäftigung mit der MDA auch dann lohnt, wenn sie noch lange davon entfernt ist „Out-of-the-box" verwendet werden zu können, also eine pragmatische Sicht geliefert wird. Bevor wir allerdings diese, in der Überschrift versprochenen pragmatischen Sichten liefern können, wollen wir dem in letzter Zeit fast überstrapazierten Begriff des Pragmatismus innerhalb der IT Leben einhauchen, und versuchen deutlich zu machen, was (zumindest in dieser Publikation) darunter verstanden werden kann.

Kernsätze des Pragmatismus

Als Kernsätze des Pragmatismus gelten laut W. James, dessen Buch *Pragmatism* im Jahr 1907 zu einer der Gründungsschriften des modernen Pragmatismus innerhalb der Philosophie gezählt wird, folgende Aussagen:

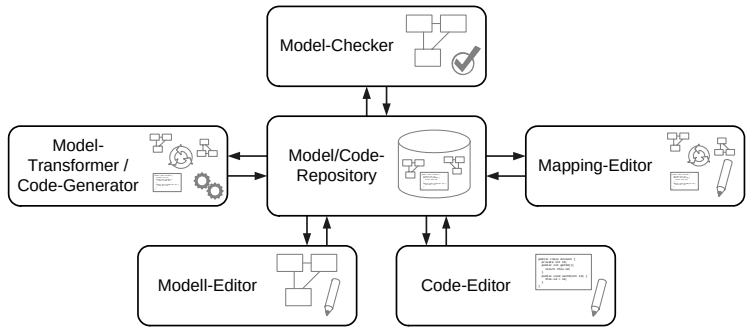
- *„Wahr ist das, was sich durch seine praktischen Konsequenzen bewährt."* und
- *„Jede Tätigkeit ist darauf hin zu überprüfen, welche Erfolge sie den sich ändernden und spezifischen Interessen aufweisen."* [Pragmatism]

Für uns heißt das, dass wir das zunächst einmal rein theoretische Paradigma der Model-Driven Architecture als Gedankengebilde der OMG darauf untersuchen müssen, ob es praktisch durchführbar ist und – nicht zu vergessen – ob ihre Anwendung einen Vorteil in der täglichen Praxis der Softwareentwicklung darstellt.

Pragmatische Sicht bedeutet dann: „Was ist *jetzt* konkret umsetzbar und was bringt es mir? Was muss/kann gegebenenfalls weggelassen werden?“. Mit anderen Worten: „Was ist schon jetzt durch die Anwendung von Teilen der MDA-Kette erreichbar?“ Das sind die Fragen, die wir im Laufe des Buches versuchen werden zu beantworten und zu deren Beantwortung wir an dieser Stelle das Fundament legen wollen.

Versuchen wir eine – vorerst hypothetische – Liste aufzustellen, die die Werkzeuge enthält, die wir uns für die Durchführung des Ansatzes wünschen würden (Abb. 2.18).
Ohne zu sehr auf Details einzugehen, kommen dem Leser mit Sicherheit viele Arten von Werkzeugen bekannt vor. Wichtigste Vertreter der bekannten Spezies dürften der *Code-Editor* sowie der *Modell-Editor* sein, deren Anwendung in der heutigen Software-Entwicklung bereits zum Standard gezählt werden darf. Auch Teilfunktionalitäten der anderen Komponenten sind vielleicht bereits aus anderen Werkzeugen bekannt, wie die seit Auftreten der ersten Komponenten-Frameworks – beispielsweise Java EE/EJB – in Mode gekommenen *Code-Generatoren.*

Abb. 2.18
MDA Werkzeug-kasten



Mapping-Editoren – zur Beschreibung der Durchführung von Modell-Transformationen – und *Model-Repositorys*, die unter anderem zur Versionierung und der Ermöglichung von Teamarbeit an Modellen verwendet werden, und damit zur Realisierung der MDA dringend benötigte Werkzeuge darstellen. Zusammen mit den *Modell-Transformatoren* selber sind sie allerdings, wenn überhaupt

vorhanden, wesentlich schwieriger verfügbar und die beteiligten Datenformate und Vorgehensweisen oftmals hochgradig proprietär.

Viele der Werkzeuge auf der Wunschliste fehlen also aktuell noch. In der Praxis bedeutet dies, dass viele Ansätze (wie *MDA-light* o.ä.) versuchen, sich der OMG-Vision der MDA zu nähern und sich dabei hauptsächlich auf die erstgenannte Gruppe der breits verfügbaren Werkzeuge stützen.

2.4.1 MDA-light

Im Falle des *MDA-light* bedeutet das konkret:

- Modellierung des PIMs mit einem UML-Modellierungswerkzeug der Wahl,
- Anreicherung des Modells mit zusätzlicher Information zur Code-Generierung,
- Automatische Generierung des Code-Rahmens aus dem Modell,
- einfügen der fehlenden Business-Logik in das generierte Gerüst,
- weiteres Iterieren über die zuvor genannten Arbeitsschritte.

Nachteile

Unserer Meinung nach lässt dieser Ansatz viele wichtige – vielleicht sogar die wichtigsten – Vorteile der MDA unbeachtet. An dieser Stelle sei beispielhaft die Verwässerung des PIM durch technische Details genannt, die an dieser Stelle die Verständlichkeit des Modells mindert sowie eine Teilung von Technik und Domänenwissen nicht ermöglicht – eines der Hauptanliegen der MDA.

…und Vorteile

Der nicht wegzudiskutierende Vorteil, den MDA-light bietet, ist jedoch ebenfalls gewichtig: Er ist bereits umgesetzt und nicht nur blanke Theorie, wie die meisten aktuellen Abhandlungen zum Thema MDA.

2.4.2 Warum jetzt?

Wenn allerdings noch ein so weiter Weg zur Realisierung der MDA-Vision zu gehen ist, stellt sich die legitime Frage, warum wir uns bereits jetzt mit der Materie so eingehend befassen sollten. Wenn es noch ein paar Jahre dauert, bis MDA zum akzeptierten Standard heranwächst – und noch nicht einmal das ist ja 100%ig sicher – warum auch nur einen Gedanken an die Sache verschwenden?

Wir sind durchaus der Meinung, dass sich eine intensive Beschäftigung mit der Materie lohnt. Das prinzipielle Proof-of-Concept zum „Beweis“, dass sich bereits jetzt Vorteile aus ersten Gehversuchen

der MDA ziehen lassen, wollen wir im zweiten Teil dieses Buches erbringen.

Eine Auswahl der wichtigsten weiteren Gründe soll im Folgenden vorgestellt werden. Dabei wollen wir zuerst die Rolle des Advocatus Diaboli einnehmen und eines der am häufigsten genannten Gegenargumente anführen:

„Das ist doch alter Wein in neuen Schläuchen!"

Dieser Einwand stützt sich auf die richtige Beobachtung, dass die MDA nichts Neues enthalte und lediglich bereits bekannte (und wenig erfolgreiche) Konzepte (im Wesentlichen die Code-Generierung innerhalb einer komplett integrierten Entwicklungsumgebung à la CASE) aufwärme.

Dass es ein Zeichen falscher Interpretation ist, den Ansatz auf eine Generierung von Quellcode bzw. die Ablösung von Quellcode durch reine Modellierung zu reduzieren, sollte bereits aus dem vorhergegangenen Text deutlich geworden sein. Zur Behauptung „Alles ist schon einmal da gewesen!" kann man sagen: „Stimmt!".

In der Tat sind viele Ideen bereits angedacht oder sogar Versuche einer technischen Realisierung unternommen worden. Eine kleine Auswahl dieser „Wurzeln" ist Ihnen aus den vorangegangenen Abschnitten bekannt. Auch ist richtig, dass viele dieser Ideen nicht den gewünschten Erfolg bzw. die verdiente Beachtung innerhalb der Software-Welt erhalten haben.

Unserer Ansicht nach lag dies aber weniger an der Unzulänglichkeit der unterliegenden Ideen und Konzepte, sondern vor allen Dingen am Fehlen dringend benötigter Standards, was es unmöglich machte, Ansätze wie *Executable UML*, *Design Patterns*, *Model-Driven Development*, *Generative Programming* usw. zu koordinieren und ein Zusammenspiel der für die einzelnen Ansätze geschaffenen Werkzeuge und Konzepte zu ermöglichen. Genau dieses Zusammenspiel durch die Schaffung von Standards zu ermöglichen und zu orchestrieren ist jedoch Hauptanliegen der MDA.

Die notwendigen Standards und technologischen Ökosysteme sind nun vorhanden

Die Zeit, einen zweiten Anlauf zu unternehmen, die bereits existierenden Ideen zu integrieren und als Ganzes nutzbar zu machen, ist unserer Meinung nach mit der Schaffung und Verfeinerung vieler neuen Standards und der Finalisierung der UML 2.x gekommen.

Offene, frei verfügbare Entwicklungsumgebungen, die durch ihren Framework-Charakter die Erstellung von leicht integrierbaren Komponenten (in Form von Plug-ins) ermöglichen – bekannte

Beispiele sind hier Eclipse [Eclipse] und Netbeans [Netbeans] – werden bereits genutzt, um einige Teilaspekte der MDA zu realisieren (z. B. [AndroMDA]) und laden dazu ein, die fehlenden Komponenten zu ergänzen bzw. an die eigenen Bedürfnisse anzupassen. Sie werden damit zu genau dem *Ökosystem*, das bei früheren Versuchen der Umsetzung gefehlt hat und in denen jetzt die Realisierung der vorgestellten Ideen gelingen kann.

Es gibt jedoch noch einen weiteren Grund, der die Beschäftigung mit der MDA zum interessanten Unterfangen gelingen lässt:

Die heißen Eisen des modernen Software-Engineering

Die bisher erwähnten Standards und Werkzeuge zählen allesamt zu den „heißen Eisen" der aktuellen Software-Entwicklung und machen nur einen Bruchteil der insgesamt an der MDA-Vision beteiligten Technologien aus.

Eine Beschäftigung mit ihnen lohnt sich unserer Meinung nach deshalb selbst dann, wenn man nicht die Einführung des MDA-Prozesses in das eigene Unternehmen in Erwägung zieht, sondern „nur" über die aktuellen Trends und Möglichkeiten informiert bleiben will. Pragmatismus heißt auch, sich unvoreingenommen alle Möglichkeiten anzuschauen, und sich dann die Sahneteilchen vom Buffet zu nehmen, die in die aktuelle Situation passen. Um dies tun zu können muss man sie allerdings auch kennen …

2.5 Also …

Egal, ob Sie nach der Lektüre dieses Buches ein Anhänger oder vielleicht einer der größten Kritiker des MDA-Ansatzes sein werden, Sie werden dies auf jeden Fall auf viel soliderer Basis tun, als viele der Diskussionsteilnehmer im Moment. Was Sie auf jeden Fall mitnehmen werden, sind Einsichten in viele spannende und interessante Ideen und Technologien, die schon für sich allein genommen eine Bereicherung ihres Software-Entwicklungs-Zyklus sein werden.

In diesem Sinne: Viel Spaß beim Weiterlesen!

Literatur

[AIS77] Christopher Alexander, Sara Ishikawa und Murray Silverstein: *A Pattern Language – Towns, Buildings, Construction*. Oxford University Press, 1977.

[AN04] Jim Arlow und Ila Neustadt: *Enterprise Patterns and MDA - Building Better Software with Archetype Patterns an UML*. Addison-Wesley, 2004.

[AndroMDA] AndroMDA.org: *Components quickly with AndroMDA*.
http://www.andromda.org/ (letzter Abruf Mai 2005)

[ASU86] Alfred V. Aho, Ravi Sethi und Jeffrey D. Ullman: *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bau93] Friedrich L. Bauer: *Software Engineering – wie es begann*. In: Informatik-Spektrum 16, S. 259-260, Springer, Februar 1993.

[BC87] Kent Beck und Ward Cunningham: *Using Pattern Languages for Object-Oriented Programs*. In: Technical Report CR-87-43, Tektronix, Inc., September 17, 1987.

[BKPS04] Günter Böckle, Peter Knauber, Klaus Pohl und Klaus Schmid (Hrsg.): *Software-Produktlinien – Methoden, Einführung und Praxis*. dpunkt, 2004.

[Boe76] Barry W. Boehm: *Software Engineering*. In: IEEE Transactions on Computers, Volume 25, Number 12, S. 1226-1241, IEEE Computer Society, Dezember 1976.

[Boo93] Grady Booch: *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 1993.

[CCM] Object Management Group (OMG): *CORBA Component Model*.
http://www.omg.org/technology/documents/formal/components.htm (letzter Abruf Mai 2005)

[CE00] Krysztof Czarnecki und Ulrich W. Eisenecker: *Generative Programming – Methods, Tools and Applications*. Addison-Wesley, 2000.

[CGN] Code Generation Network: *Code Generation Information for the Pragmatic Engineer*.
http://www.codegeneration.net/ (letzter Abruf Mai 2005)

[COWO00] COMPUTERWOCHE.de: *Java-Projekt Cheops in den Sand gesetzt*.
http://www.computerwoche.de/index.cfm?artid=17266 (letzter Abruf Mai 2005)

[CT03] Detlef Borchers: *Verursacherbedingt verspätet – Das „fortschrittlichste Mautsystem der Welt" und die Realität*. In: c´t 22/2003, Seite 92, Heise Zeitschriften Verlag, 2003.

[DBC] Wikipedia: *Design by Contract*.
http://en.wikipedia.org/wiki/Design_by_contract (letzter Abruf Juli 2005)

[DDH72] Ole-Johan Dahl, Edsger W. Dijkstra und C. A. R. Hoare: *Structed Programming*. Academic Press, 1972.

[DE] The Software Engineering Institute (SEI): *Domain Engineering: A Model-Based Approach*.
http://www.sei.cmu.edu/domain-engineering/ (letzter Abruf Mai 2005)

[DeM78] Tom DeMarco: *Structured Analysis and System Specification*. Yourdon Press, 1978.

[Eclipse] Eclipse.org: *Eclipse.org Main Page*.
http://www.eclipse.org/ (letzter Abruf Mai 2005)

[EJB] Sun Microsystems: *Enterprise JavaBeans Technology*.
http://java.sun.com/products/ejb/ (letzter Abruf Mai 2005)

[Fow96] Martin Fowler: *Analysis Patterns – Reusable Object Models*. Addison-Wesley, 1996.

[Gla02] Robert L. Glass: *Facts and Fallacies of Software Engineering*. Addison-Wesley, 2002.

[GSCK04] Jack Greenfield, Keith Short, Steve Cook und Stuart Kent: *Software Factories – Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, 2004.

[HO04] heise online: *Bericht – Software-Projekt für Finanzämter gescheitert*.
http://www.heise.de/newsticker/meldung/48843 (letzter Abruf Mai 2005)

[HS99] Peter Herzum und Oliver Sims: *Business Component Factory – A Comprehensive Overview of Component-Based Development for the Enterprise*. John Wiley & Sons, 1999.

[Hub01] Richard Hubert: *Convergent Architecture – Building Model-Driven J2EE Systems with UML*. John Wiley & Sons, 2001.

[HUTN] Object Management Group (OMG): *Human-Usable Textual Notation (HUTN) v1.0*.
http://www.omg.org/docs/formal/04-08-01.pdf (letzter Abruf Mai 2005)

[IEEE1471] IEEE Computer Society: *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems [IEEE Std 1471-2000]*. IEEE Computer Society, 2000.

[JavaEE] Sun Microsystems: *Java Platform, Enterprise Edition (Java EE)*.
http://java.sun.com/j2ee/ (letzter Abruf Mai 2005)

[JBCR01] Jim Johnson, Karen D. Boucher, Kyle Connors und James Robinson: *Collaborating on Project Success*. In: Software Magazine, Wiesner Publishing, Februar/März 2001.

[JBR98] Ivar Jacobson, Grady Booch und James Rumbaugh: *The Unified Software Development Process*. Addison-Wesley, 1998.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina, Videira Lopes, Jean-Marc Loingtier und John Irwin: *Aspect-Oriented Programming*. In: European Conference on Object-Oriented Programming, Springer, 1997.

[MB02] Stephen J. Mellor und Marc J. Balcer: *Executable UML – A Foundation for Model Driven Architecture*. Addison-Wesley, 2002.

[MDAATP] Object Management Group (OMG): *Model Driven Architecture – A Technical Perspective*. http://www.omg.org/cgi-bin/apps/doc?ormsc/01-07-01.pdf (letzter Abruf Mai 2005)

[MDAGuide] Object Management Group (OMG): *MDA Guide Version 1.0.1*. http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf (letzter Abruf Mai 2005)

[MW] Wikipedia: *Middleware*. http://de.wikipedia.org/wiki/Middleware (letzter Abruf Mai 2005)

[NetBeans] NetBeans.org: *Welcome to NetBeans*. http://www.netbeans.org/ (letzter Abruf Mai 2005)

[NRB76] Peter Naur, Brian Randell und John N. Buxton (Hrsg.): *Software Engineering – Concepts and Techniques – Proceedings of the NATO Conferences*. Petrocelli/Charter, 1976.

[Pragmatism] Wikipedia: *Pragmatism*. http://en.wikipedia.org/wiki/Pragmatism (letzter Abruf Mai 2005)

[Qui94] Klaus Quibeldey-Cirkel: *Paradigmenwechsel im Software-Engineering – Auf dem Weg zu objektorientierten Weltmodellen*. In: Softwaretechnik Trends 14/1, S. 47–58, Gesellschaft für Informatik, 1994.

[Ra06] Noorg Raglof: *Pair Ranting – The next step in Hyperagility*. Submitted paper for the Waterfall 2006 Confernce. The Waterfall Alliance, 2006. http://www.waterfall2006.com/ (letzter Abruf April 2006)

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy und William Lorensen: *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[SDL] ITU-T: *Specification and Description Language (SDL) [ITU-T-Z.100]*. ITU-T, 2002.

[SF03] Howard Smith und Peter Fingar: *IT Doesn't Matter – Business Processes Do – A Critical Analysis of Nicholas Carr's I.T. Article in the Harvard Business Review*. Meghan-Kiffer, 2003.

[SG03] The Standish Group: *Latest Standish Group CHAOS Report Shows Project Success Rates Have Improved by 50%*.
http://www.standishgroup.com/press/article.php?id=2 (letzter Abruf Mai 2005)

[SGW94] Bran Selic, Garth Gullekson und Paul T. Ward: *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.

[SPL03] Robert C. Seacord, Daniel Plakosh und Grace A. Lewis: *Modernizing Legacy Systems – Software Technologies, Engineering Processes, and Business Practices*. Addison Wesley, 2003.

[Tay95] David A. Taylor: *Business Engineering with Object Technology*. John Wiley & Sons, 1995.

[UML] Object Management Group (OMG): *UML Resource Page*.
http://www.omg.org/uml/ (letzter Abruf Mai 2005)

[UPfEDOC] Object Management Group (OMG): *UML Profile For Enterprise Distributed Object Computing (EDOC)*.
http://www.omg.org/technology/documents/formal/edoc.htm (letzter Abruf Mai 2005)

[UPfEJB] Java Community Process: *JSR 26 – UML/EJB Mapping Specification*.
http://www.jcp.org/aboutJava/communityprocess/review/jsr026/ (letzter Abruf Mai 2005)

[YC79] Edward Yourdon und Larry L. Constantine: *Structured Design – Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice Hall, 1979.

[You02] Edward Yourdon: *Managing High-Intensity Internet Projects*. Prentice Hall, 2002.

3 Modellierung

„It's a bunch of shapes connected by lines."

Dilbert

Die Integration von Modellen in den Entwicklungsprozess ist zentrales Konzept modellgetriebener Ansätze und insbesondere der Model-Driven Architecture zur Beherrschung von Komplexität. Im folgenden Kapitel wollen wir uns daher die Zeit nehmen und die notwendigen Grundlagen schaffen, um das unterliegende Potenzial voll auszunutzen. Dabei wird sowohl auf die theoretischen Grundlagen eingegangen, als auch praktische Umsetzung mittels Unified Modeling Language (UML) begonnen.

3.1 Grundlagen der Modellierung

Modelle entstehen durch die Beschreibung von Systemen. Um die Behauptung nachvollziehen zu können, dies helfe bei der Bewältigung von Komplexität, ist es notwendig, die Begriffe „Modell" und „System" genauer zu definieren.

IEEE 1471 definiert den Begriff *System* wie folgt:
Ein *System* ist ein aus Teilen zusammengesetztes und strukturiertes Ganzes. Es hat eine Funktion, erfüllt einen Zweck und verfügt über eine Architektur.

Dies bedeutet, dass so ziemlich alles in unserem Alltag als System aufgefasst werden kann: materielle Dinge (z. B. das Auto, das aus seinen Einzelteilen wie Räder, Lenkrad usw. besteht), aber auch mehr konzeptionelle Einheiten wie Software (Module, Interfaces, EJBs, …) oder Geschäftsprozesse (Aktivitäten, Workflows) usw. Ein *Modell* letztendlich beschreibt ein System in geeigneter Form.

Wie sich bald herausstellen wird, stellt das Herausfinden der „geeigneten Form“ dabei keine triviale Aufgabe dar.

Die Darstellung des Modells kann auf verschiedenste Art erfolgen: grafisch, textuell oder durch Hybride, also Mischformen dieser beiden. Handelt es sich um im Wesentlichen visuell dargestellte Modelle, so werden diese im Allgemeinen als *Diagramme* bezeichnet.

Jede dieser Arten wiederum kann formalen oder skizzenhaften Charakter besitzen. Was dies genau bedeutet, wird im Verlaufe dieses Kapitels noch näher beleuchtet.

Systeme und Subsysteme

Wichtiges Konzept bei der Definition von Systemen ist das der *Fraktalität* (in etwa: stark gegliederte Selbstähnlichkeit). Schlicht ausgedrückt heißt das lediglich, dass ein System seinerseits wieder aus mehreren (Sub-)Systemen besteht. Diese Eigenschaft wird innerhalb der Modellierung zur vereinfachenden Darstellung des Systems ausgenutzt. Durch die Aufteilung eines Systems in Subsysteme und die Darstellung der Zusammenhänge, die zwischen diesen bestehen (eine solche Darstellung wird üblicherweise auch Architekturbeschreibung eines Systems genannt), können auch hochkomplexe Sachverhalte in ihre Einzelkomponenten aufgeteilt und somit beherrschbar gemacht werden. Entscheidend für den Erfolg eines solchen Unterfangens ist die Granularität (in etwa der Detaillierungsgrad) der dabei gebildeten und als ein Teil des Modells dargestellten *Module* (Einheiten). Je mehr Einzelheiten, die für den gewünschten Überblick oder für das betrachtete Modul nicht von Bedeutung sind, in der Darstellung weggelassen bzw. vereinfacht werden, desto übersichtlicher (und damit hoffentlich auch verständlicher) wird die Beschreibung.

Ein beliebtes Kriterium zum Schnitt der gebildeten Module ist die Autonomie ihrer Aufgabenbereiche. Elemente des Systems, die zusammen eine Aufgabe oder Anforderung erfüllen, bilden ein Modul und werden gemeinsam dargestellt. Dabei muss es sich nicht um rein funktionelle Aufgabenbereiche handeln, wie z. B. die gemeinsame Realisierung einer Log-In Funktionalität. Vielmehr sind auch die nichtfunktionalen Anforderungen mit einzubeziehen, die das System zu erfüllen hat. Typische Beispiele hierzu sind die Darstellung der technischen Infrastruktur zur Erfüllung von Sicherheits- und Verfügbarkeitsanforderungen oder die Modellierung transaktioneller Klammern in Geschäftsprozessen.

Separation of Concerns

Ein oft genannter Begriff in der Modulbildung unter diesem Gesichtspunkt ist der des *Separation of Concerns (engl.)*, ein Begriff, dessen Ursprünge, obwohl dort nicht so genannt, auf eine Publika-

tion von David L. Parnas zurückgeführt werden, die schon 1972 erschienen ist [Par72]. Auch in diesem Kontext meinen Concerns spezifische Anforderungen oder Bedingungen, die erfüllt werden müssen, um das die Anforderungen an das System zu erfüllen. Wir wollen Concern im Folgenden mit *Aspekt* (eines Systems) übersetzen. Der Begriff taucht in ähnlicher Bedeutung auch im Bereich der Aspekt-Orientierten Programmierung (AOP) [ORe04] [Lad03] auf. Hier wird zurzeit vor allem eine besondere Klasse von Aspekten betrachtet: die der Systemübergreifenden. In diesem Buch soll der Begriff Aspekt jedoch abstrakt für jedwede Eigenschaft bzw. einer kohärenten Menge von Eigenschaften eines Systems verwendet werden, die für aktuelle Betrachtung interessant ist. Dies können so unterschiedliche Details sein wie: die Datenstrukturen einer konkreten Implementierung, die Beschreibung von (Geschäfts-)Prozessen, die Dokumentation architektureller Entscheidungen in kommentierter Diagrammform, Konfigurationswissen, Dienstequalität (engl. Quality of Service, QoS), die Verteilung technischer Komponenten usw.

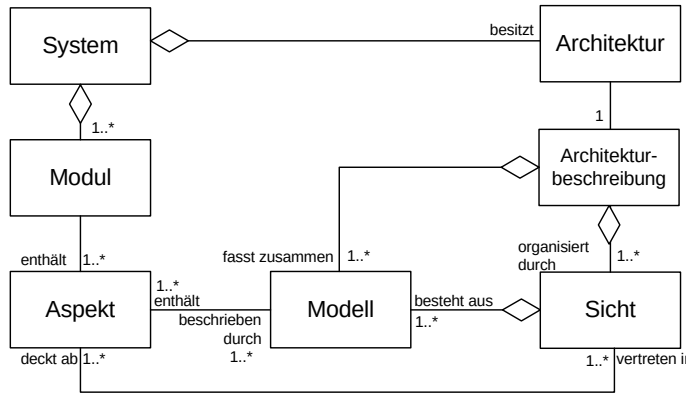
Beim Betrachten der Aufzählung fällt auf, dass Aspekte in zwei Kategorien aufgeteilt werden können: auf der einen Seite die bereits im Rahmen der AOP erwähnten systemübergreifenden Anforderungen (Qualitätssicherung, Prozesse, …), auf der anderen Seite die auf lokalen Einheiten greifenden wie z. B. die Kapselung von Datenstrukturen oder die Aufteilung in ausführbare und verteilbare Einheiten. Während lokale Aspekte bereits durch Techniken wie der Objektorientierung (Datentypen, funktionale Modulbildung) oder der komponentenbasierten Entwicklung ganz gut in den Griff bekommen worden sind, ist die Beherrschung der Querschnittseigenschaften eines Systems – man nennt systemübergreifende Aspekte auch Querschnittseigenschaften – noch relatives Neuland. Hier konkurrieren zurzeit eine Vielzahl von Ansätzen, die versuchen die Software-Entwicklung in diesem Bereich effizienter zu gestalten. Technologien wie das bereits erwähnte AOP oder das *Generative Programming* (vgl. [GP] bzw. [CE00]) sind zwei der bekannteren Vertreter, die hier Fortschritte zu vermelden haben, widmen sich zurzeit jedoch vornehmlich dem technisch-funktionalen Bereich. Immer wieder auftauchendes Beispiel hierfür ist die Querschnittsaufgabe Logging, die quasi das *Hello World* der systemübergreifenden Aspekte darstellt. Klare Defizite besitzen diese Ansätze jedoch in der modelltechnischen Erfassung abstrakterer Ebenen, wie beispielsweise der Geschäftsprozessmodellierung o. Ä. sowie der produktiven Integration dieser Aspekte in den Entwicklungsprozess.

MDA macht dieses Wissen abbild- und damit auch im Entwicklungsprozess verfügbar, indem es Modellierungsmöglichkeiten für beide Arten von Aspekten zur Verfügung stellt. Die vollständige Zerlegung des Systems in Module – gemäß seiner Aspekte – und die Erstellung von Modellen zur Abbildung dieser Aspekte durch die Bereitstellung geeigneter Notationselemente wird ebenso möglich wie die Nutzung dieser Modelle zur Wiederverwendung, Optimierung usw. - kurz: die vollständige aktive Integration in den Entwicklungsprozess.

Sichten, Architektur, System

Bevor wir fortfahren ordnen wir noch einmal kurz das eben Gelesene: Eine kohärente Menge von Modellen bzw. Teilmengen eines Gesamtmodells, die einen oder mehrere Aspekte eines Systems beschreiben und für den gedachten Verwendungszweck irrelevante Details ausblenden, wird innerhalb der MDA als *Sicht (engl. View)* auf dieses System bezeichnet. Die Menge aller der Sichten zusammen mit eventuell notwendiger Dokumentation beschreibt die *Architektur* eines Systems. Die folgende Abbildung (Abb. 3.1) verdeutlicht die Zusammenhänge noch einmal:

Abb. 3.1 Begriffzusammenhänge



Im Prozess der *Modellierung* wird ein System in ein oder mehrere *Module* zergliedert. Idealerweise umfassen diese genau einen *Aspekt* des Systems und werden in jeweils einem (Teil-)Modell beschrieben. Eine *Sicht* fasst eine kohärente Menge dieser Modelle zusammen. Die Menge aller Sichten wiederum bilden zusammen mit eventuell notwendiger Zusatzdokumentation das *Gesamtmodell* oder die *Architekturbeschreibung* der betrachteten *Systemarchitektur*.

Prinzip der Abstraktion

Will man die geschaffenen Module weiter zergliedern, „zoomt“ man in das Modell hinein, das Aspekte beschreibt, die detailliert werden sollen und beschreibt diese Einzelheiten wiederum in geeigneten Modellen. So werden auch komplexe Systeme nach dem *„teile-und-herrsche“*-Prinzip verwalt- und darstellbar. Ein „Herauszoomen“ entspricht dann folgerichtig der Vernachlässigung von Einzelheiten in den jeweiligen Modellen. So ist es möglich, auch das „große Ganze“ nicht aus den Augen zu verlieren. Das Prinzip des „geeigneten Weglassens von Einzelheiten“ nennt man *Abstraktion*. Je abstrakter ein Modell, desto grobgranularer wird es (es werden also mehr Einzelheiten „vergessen“, die Komplexität (des Modells) sinkt). Den Grad der Abstraktion, auf der sich ein Modell befindet, nennt man *Abstraktionsniveau*.

Das Abstrahieren einer Systemsicht auf ein geeignetes Niveau ist ein keineswegs triviales Problem, dessen Komplexität stark von dem abhängt, was das Modell dem Betrachter vermitteln soll. Der Vorgang der Abstraktion verlangt sowohl eine gewisse Erfahrung in der Sache selbst als auch das nötige fachliche Wissen über das System, das modelliert werden soll. *Last, not least* werden die handwerklichen Fähigkeiten verlangt, aus den gewonnenen Erkenntnissen auch ein fehlerfreies Modell im Sinne der verwendeten Beschreibungssprache erstellen zu können. Es bestätigt sich hier die alte Erkenntnis, dass nicht die Informationsgewinnung die Kosten verursacht, sondern das Loswerden derselben. Hilfreich zum Erreichen dieses Ziels sind Techniken wie Brainstorming Sitzungen und zielgerichtet *Trial-and-Error* Verfahren, die alle das Eine gemeinsam haben: Sie führen notwendigerweise nur inkrementell zu einem Ergebnis. Es gibt also niemals *das* endgültig richtige (Gesamt-)Modell, sondern vielmehr ein mögliches Ergebnis, an dem nach und nach Verbesserungen vorgenommen werden, wenn das Verständnis des Problems wächst oder Unzulänglichkeiten des ursprünglichen Modells eine Nachbesserung nötig macht. Mit der „Korrektheit“ des Modells nimmt gleichzeitig das gemeinsame Verständnis des Systems unter denjenigen zu, die an der Modellierung beteiligt sind. Nicht selten werden erst bei expliziter Ausmodellierung bestimmter Aspekte Lücken oder Missverständnisse aufgedeckt, die sonst unentdeckt bleiben würden.

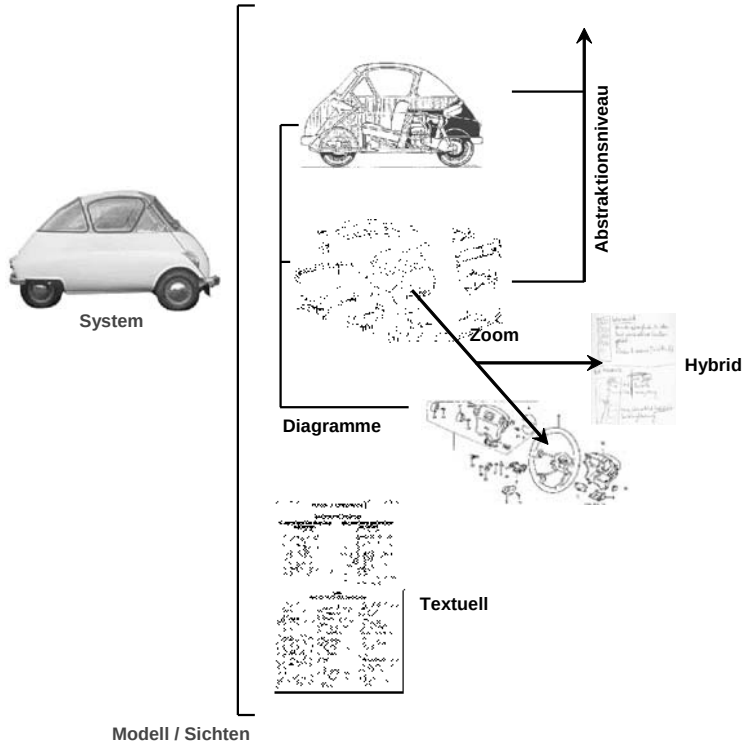
Kann aus der Gesamtheit aller so entstandenen Beschreibungen das Ursprungssystem vollständig rekonstruiert werden, nennt man das Ergebnis dieser inkrementellen Arbeit – die Menge aller erstellten Sichten – die *Spezifikation* eines Systems. Die erste Entwicklungsphase, die mit der Schaffung einer solchen Spezifikation endet, heißt *Analyse*. Im Laufe der Evolution der Softwareentwicklung sind viele

Vorgehensmodelle zur Durchführung von Analysen vorgeschlagen worden, einige davon werden wir im Verlaufe des Buches noch näher kennen lernen und auf ihre Eignung für die MDA untersuchen.

Ein Beispiel

Nach der trockenen Theorie nun noch etwas „Handfestes“ (Abb. 3.2):

Abb. 3.2 Beispielsystem



Wir sehen auf der linken Seite ein Automobil des Typs Isetta, das im folgenden Beispiel als betrachtetes System fungieren soll. Auf der rechten Seite der Abbildung finden wir Beschreibungen, die uns unterschiedliche Sichten auf unser Fortbewegungsmittel liefern: eine grobe Übersicht in Form einer Zeichnung oben, die lediglich als schematische Darstellung dienen kann sowie eine detailliertere technische darunter, die auch die Zerlegung des Autos in seine Bestandteile darstellt. Zu dieser Beschreibung existieren zusätzliche Dokumente, die ausgewählte Teile näher erläutern (Zeichnung des Armaturenbretts), aber auch kommentierte Skizzen, die uns nähere Informationen über diese Einzelteile wie Abmessungen, Einbau usw. vermitteln (*Hybride*). Letztlich noch eine rein textuelle Be-

schreibung, die ganz ohne graphische Elemente auskommt, jedoch trotzdem ein Modell unseres Autos darstellt, da es wichtige Details (natürlichsprachlich (Prosa) oder in einer speziellen Spezifikationssprache) vermittelt.
Sind unsere Modelle – sprich unsere Spezifikation – genau genug, den Fahrzeugtyp in allen relevanten Einzelheiten zu beschreiben, ist jedermann, der über die nötige Fähigkeit verfügt die Pläne zu lesen und die benötigten Materialien zu beschaffen, in der Lage seine eigene Isetta zu bauen (die notwendigen Fertigkeiten vorausgesetzt).

Hat der Wagen nach gewissenhaftem Befolgen der Anweisungen nur zwei Räder oder fehlt das Lenkrad, gibt es prinzipiell drei Möglichkeiten, die zu der Misere geführt haben können:

- Die gelieferten Pläne waren fehlerhaft,
- der Monteur hat sie falsch interpretiert,
- sie enthielten schlicht nicht die notwendigen Details, die eine korrekte Montage des Fahrzeugs ermöglicht hätten.

Da ein Auto ohne Lenkrad ähnlich erfreulich ist wie ein Programm zur Textverarbeitung ohne Speicher-Button, gilt es folglich möglichst die Erstellung ungenauer Modelle zu vermeiden. Was aber macht Modelle „genau genug“? Und wie sieht so ein „geeignetes“ Modell aus? Wie erwähnt, spielt bei der Findung des geeigneten Abstraktionsgrades der Zweck, für den ein Modell erstellt wird, eine tragende Rolle. „Zu ungenau“ bedeutet, dass zu viele Informationen weggelassen, also zu sehr vom realen System abstrahiert wurde, um die Handlung, die eigentlich mithilfe des Modells durchgeführt werden soll, erfolgreich durchzuführen.

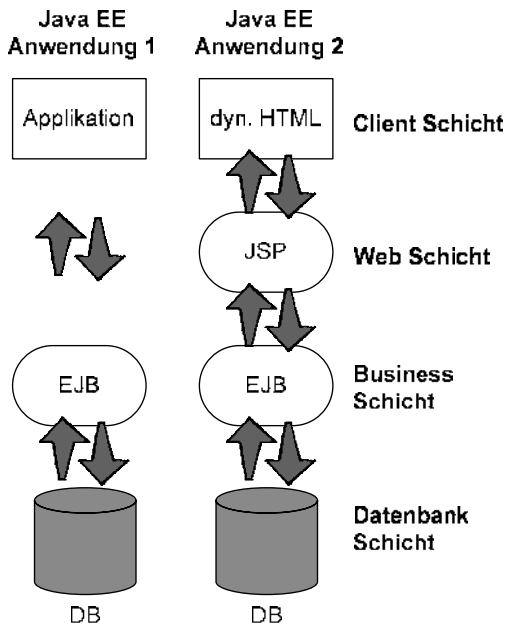
3.1.1 Sketch-Modelle

Um einem menschlichen Betrachter lediglich einen groben Überblick über die Struktur, beispielsweise den Aufbau, eines Systems zu vermitteln und so sein Verständnis eines Sachverhaltes zu vertiefen, sind grobgranulare Beschreibungen völlig ausreichend. Man nennt diese Art von Modellen *Sketch-Modelle*, um ihren skizzenhaften Charakter hervorzuheben [Fow04a].

Sketch-Modelle sind vorwiegend Diagramme, die keine Legende erfordern, da sie selbsterklärend sind oder sich deren Bedeutung aus dem mitgelieferten Kontext (mündliche Erklärung, Begleittext) ergibt. Als Beispiel eines solchen Modells dient hier Abb. 3.3, die

dem Java Tutorial [CWH00] entliehen ist und eine typische Struktur von Java EE Architekturen zeigen soll.

Abb. 3.3 Beispiel für ein Sketch Modell



Auch ohne Kenntnisse der verwendeten Technologien ist – ein wenig Erfahrung in der Softwareentwicklung vorausgesetzt – Folgendes aus dem Diagramm zu erkennen:

- Es sind zwei Anwendungen dargestellt,
- die Anwendungen sind in Schichten aufgeteilt,
- die Schichten der Anwendungen sowie die darin enthaltenen Komponenten haben Bezeichnungen und spezielle Formen/Farben,
- die Komponenten an angrenzenden Schichten einer Anwendung sind durch Pfeile miteinander verbunden.

Damit ist der Informationsgehalt der Zeichnung für den unvorbereiteten Betrachter allerdings auch schon erschöpft. Er erfährt nichts über die Bedeutung der unterschiedlichen Farben und Formen oder darüber, wie die doppelköpfigen Pfeile zu interpretieren sind. Anscheinend passiert irgendetwas Wichtiges zwischen den Schichten, was genau das ist bleibt allerdings im Dunkeln.

Dieses Modell allein für sich ist damit keine adäquate Beschreibung des unterliegenden Systems. Existiert jedoch ein begleitender Text oder eine mündliche Erklärung, wie beispielsweise ein Vortrag, und dient die Zeichnung der Visualisierung eines in diesem Text genau beschriebenen Sachverhaltes, stellt sie also nur einen Teil des Systems dar, und wird sie durch das textuelle Modell ergänzt, reicht der Informationsgehalt völlig aus.

Sketch-Modelle dienen oft zum Zwecke der zwischenmenschlichen Kommunikation. Zu Problemen kommt es bei der Verwendung von Sketch Modellen nur, wenn Informationen aus diesen Modellen gezogen werden sollen ohne

- nachfragen zu können, was der Modellierende an unklaren Stellen gemeint hat, oder ohne
- fehlende Informationen ergänzen zu können, die benötigt aber nicht im Modell dargestellt werden.

Einfaches Nachfragen ist sicherlich die bequemste Art Ungereimtheiten aufzulösen bzw. fehlende Details zu beschaffen. Ist der Autor des Dokumentes aber nicht erreichbar, kann also der Betrachter keine Fragen stellen, oder ist der Betrachter gar ein Generator, der keine erklärende Prosa lesen kann aber ein Datenbankschema erzeugen soll, so muss wohl eine andere Möglichkeit gewählt werden, um unsere Anforderungen an ein gutes Modell zu erfüllen.

3.1.2 Formale Modelle

Die mögliche Antwort auf die eben aufgezeigten Probleme sind formale Modelle. „Formal“ soll hier im Sinne von „deterministisch auswertbar“ gebraucht werden. Unterschiedliche Betrachter, die die verwendeten Regeln zur Modellierung kennen, müssen bei Betrachtung der Modelle ohne weitere Erklärungen immer zum selben Ergebnis gelangen können. Dies erfordert die Existenz einer eindeutigen *Syntax* und einer eindeutigen *Semantik*. Was aber heißt das für unsere maschinelle Betrachterin und ihre Verständnisprobleme? Wenn ein Programm in der Lage sein soll ein Modell auszuwerten, sind zwei Schritte notwendig:

- Es muss entscheiden, ob das Modell fehlerfrei beschrieben worden ist (man nennt diesen Vorgang auch Parsen) und dann
- das Modell interpretieren, um die gewünschten Aktionen durchzuführen.

Syntax & Semantik

Zur Erfüllung der ersten Aufgabe braucht es die Information, welche Elemente (Symbole, Zeichen, Wörter) in welcher Kombination im Modell vorkommen dürfen. Man nennt diese Eigenschaften die Syntax einer (Modellierungs-)Sprache. Der zweite Schritt ist dann die Entscheidung in welcher Weise auf spezielle Kombinationen zu reagieren ist, also welcher „Sinn“, welche Bedeutung (Semantik) hinter ihnen verborgen ist. Aber woher weiß unser Programm welche Semantik hinter welcher Syntax steckt? Auch hier helfen uns Spezifikationen. Betrachten wir unser Programm als System, das eine Modellierungssprache interpretiert, so brauchen wir nur noch eine Spezifikation, die beschreibt, wie sich ein Programm wie unseres zu verhalten hat und können dann die Software nach dieser Anleitung programmieren.

Ein Werkzeug, das uns die Erstellung solcher Spezifikationen ermöglicht, ist die *Unified Modeling Language (UML)* der *Object Management Group (OMG)*. Dazu aber in Kürze mehr (siehe 3.2).

Turing-vollständige Sprachen

Um genügend Information in einem Modell unterbringen zu können, benötigt man Modellierungssprachen mit ausreichender Fähigkeit Semantik zu transportieren. Eine besondere Klasse von Sprachen dieser Art ist die der *turing-vollständigen Sprachen (engl. turing-complete)* (vgl. dazu auch [BL74]). Mit ihnen können Algorithmen beschrieben werden, die jedes (bekannte) berechenbare Problem lösen. Damit kann man mit ihnen aber auch jedes System, das ein berechenbares Problem löst (mit schlichten Worten also eine Anwendung), beschreiben. Aus dieser Eigenschaft folgt, dass alle turing-vollständigen Sprachen gleich mächtig sind. Eine mit einer solchen Sprache beschriebene Anwendung kann in der exakt selben Bedeutung auch mit jeder anderen turing-vollständigen Sprache beschrieben werden. Bekannte Beispiele für solche Sprachen sind universelle Programmiersprachen (engl. General-purpose programming languages, GPLs) wie Pascal, Java, C/C++, C#, aber auch Exemplare von denen man dies weniger vermuten würde, wie die Excel Makro Sprache oder reguläre Ausdrücke. Modellierungssprachen können ebenfalls zu dieser Klasse von Sprachen gehören. Eine Richtung, die dieses Ziel für die UML verfolgt, ist die Executable UML [MB02], von der wir an vorangegangener Stelle schon einmal gehört haben.

Aufgrund der Wichtigkeit für die weiteren Betrachtungen sollten wir die eben kennen gelernte Sichtweise noch einmal Revue passieren lassen:

In der herkömmlichen Software-Entwicklung beschreibt Quellcode (z. B. in der GPL Java verfasst) den, gemeinhin als Anwendung bezeichneten, Weg zur Lösung eines Problems. Sehen wir die Lösung des Problems als System an, das durch unsere Anwendung realisiert wird, so ist der Quellcode ein Modell dieses Systems. Java ist, wie gerade erläutert, turing-vollständig und alle Sprachen dieser Klasse sind gleich mächtig. Finden wir also eine turing-vollständige Modellierungssprache, so lassen sich auch in dieser alle denkbaren Probleme (Anwendungen) beschreiben und beliebige andere Repräsentationen des ursprünglichen Java Programms (wie beispielsweise Session Beans und der zugehörige Glue-Code) aus eben diesem Modell generieren bzw. das Modell selbst als Anwendung ausführen.

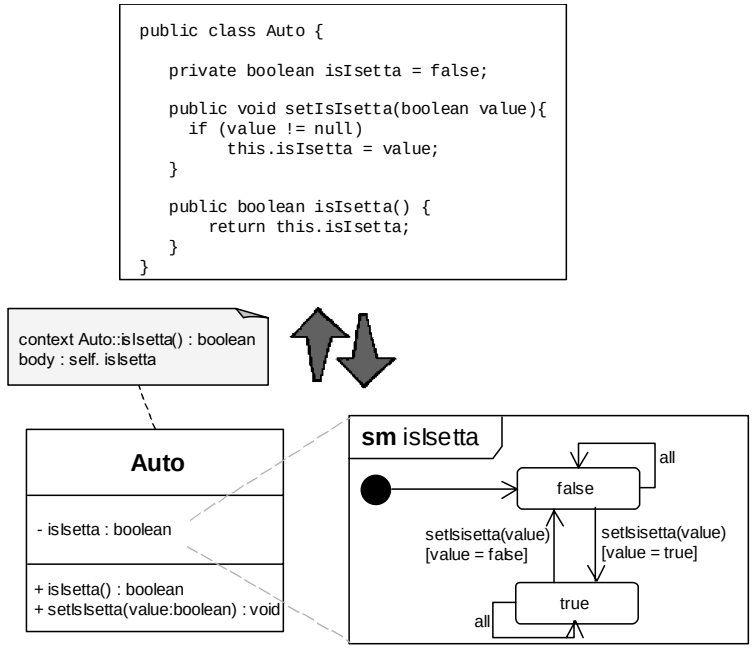


Abb. 3.4 Quellcode und grafisches Modell

Die Entscheidung, ob und wann eine solche Vorgehensweise einen Vorteil verspricht, soll vorerst frei in den Raum gestellt sein. Ein Beispiel für ein Quellcode Modell und einer graphischen Beschreibung, die denselben Sachverhalt darstellt, sehen wir in (Abb. 3.4).

Oben dargestellt der Java-Quellcode einer Klasse `Auto`, mit dem Attribut `isIsetta` und dem zugehörigen `getter/setter` Paar. Der gleiche Sachverhalt ist darunter als UML-2-Modell in Form eines Klassendiagramms dargestellt, dessen dynamisches Verhalten

zur Verdeutlichung einiger Modellierungsmöglichkeiten zum einen durch einen *Zustandsautomaten (engl. state machine)*, zum anderen durch die Anwendung der *Object Constraint Language (OCL)* (vgl. Abschnitt 3.6) dargestellt wird.

Äquivalenz von Modellen

Der Doppelpfeil bedeutet, dass eine Sicht aus der anderen erstellt werden kann, ohne zusätzliche Informationen zu benötigen. Eine Darstellung kann in die andere transformiert werden. Wenn zwei Modelle denselben Sachverhalt auf demselben Abstraktionsniveau lediglich in anderer Notation darstellen, sprechen wir von äquivalenten Modellen.

Kurzer Exkurs: Abstrakte Syntax vs. Konkrete Syntax

Hinter verschiedenen konkreten Notationen von Sprachen verbergen sich demnach mitunter gleiche bzw. gleichartige Konzepte die den gleichen oder zumindest ähnlichen Regeln gehorchen. In der angegebenen Abbildung ist beispielsweise das abstrakte Konzept der Klasse einmal mit dem Java-Schlüsselwort `class` und einmal mit dem UML-Klassensymbol in Form eines Rechtecks notiert. Des Weiteren sehen wir, dass in der Java-Notation der „Besitz“ von Attributen darüber ausgedrückt wird, dass sie innerhalb der Klassen-Klammerung (`class{ [...] }`) aufgelistet werden. In der UML-Notation werden sie in analoger Weise innerhalb einer eigenen Abteilung des Klassensymbols geführt.

Bezüglich der Regeln einer Sprache haben wir bislang nur allgemein von *der* Syntax gesprochen. Möchte man die angesprochenen Konzepte und Regeln vergleichbar machen, so erscheint es sinnvoll zwischen den abstrakten Konzepten und der Repräsentation einer Sprache zu differenzieren und eine begriffliche Trennung dafür zu finden (wir werden später noch sehen, dass diese Unterscheidung nicht nur vor diesem Hintergrund hilfreich ist). Als Termini technici haben sich dabei die *abstrakte Syntax* und die *konkrete Syntax* etabliert – sie werden wie folgt gegeneinander abgegrenzt:

Abstrakte vs. Konkrete Syntax

- Die *abstrakte Syntax* einer Sprache ist unabhängig von einer konkreten Darstellung und beschreibt wie die Konzepte einer Sprache in struktureller Hinsicht in Beziehung zueinander stehen.
- Dagegen beschreibt die *konkrete Syntax* wie diese Elemente tatsächlich (textuell und/oder grafisch) dargestellt werden können.

Auch wenn Sie damit vielleicht eine erste Idee davon bekommen haben, wie sich beispielsweise die Konzepte verschiedener Sprachen – seien sie nun textuell, grafisch oder hybrid – in Bezug setzen lassen könnten, erscheint Ihnen diese Differenzierung an dieser Stelle wohlmöglich noch esoterisch, zumal wir noch gar nicht gesehen haben, wie die abstrakte bzw. konkrete Syntax einer Sprache festgehalten werden kann. Sie wird uns im Verlaufe dieses Kapitels bzw. Buches aber noch mehrmals hilfreich sein und dann dort nach und nach Licht ins Dunkel bringen.

Brauchen wir mehr als „reinen" Quellcode?

Aber zurück zu unseren Quellcode-Überlegungen: Wenn universelle Programmiersprachen zur Beschreibung aller denkbaren Programmierprobleme geeignet sind, stellt sich die an obiger Stelle schon leise angesprochene Frage, warum wir überhaupt noch zusätzliche Modellierungssprachen brauchen. Können wir uns nicht „wie bisher" auf die direkte Erstellung von Programmen, sprich das Produzieren von Quellcode beschränken? Tatsächlich vertreten einige Kritiker innerhalb der Community die Meinung, GPLs allein reichen als formale Sprachen völlig zur Erstellung von guter Software aus. Sie lehnen die Benutzung formaler Diagramme, beispielsweise zur Kontrollflussmodellierung und daraus resultierender automatischer Generierung von Quellcode als überflüssig ab, da unübersichtlich und damit letztlich unpraktikabel. Lediglich die informale Benutzung als Gedächtnisstütze während der Analyse (also im Rahmen der erwähnten Sketch Modelle) oder zum Zwecke der Erläuterung bestimmter Details lassen sie gelten (vgl. [Fow04b] und [Amb03a]).

Um eine fundierte Antwort auf diesen Einwand liefern zu können, müssen wir eine weitere Eigenschaft in Betracht ziehen, die jedes gute Modell neben der formalen Korrektheit besitzen muss: Verständlichkeit. Hierzu ein Beispiel aus dem Alltag der Software-Entwicklung:

> *Finde den Fehler in einer Anwendung des Umfanges mehrerer hunderttausend Zeilen Quellcode, die du weder mitkonzipiert, noch -programmiert hast. P. S. Die „Anwendung" besteht eigentlich aus mehreren Einzelanwendungen, die irgendwie miteinander kommunizieren. Viel Glück!*

Intuitiv ahnt man, dass die Anwendung des Modularisierungsansatzes sowie die konsequente Erstellung einer formalen (!) Spezifikation, die nicht nur aus dem Quellcode(-modell) besteht, sondern die Vorteile adäquater Visualisierung komplexer Sachverhalte ausnutzt an dieser Stelle Vorteile bringt. Im weiteren Verlauf wollen wir

Beispiele für die konsequente Verfolgung dieser Idee liefern, um die Ahnung zur Gewissheit werden zu lassen.

Sprach-Spezialisierung

Sprachen, die prinzipiell alles mit einer Notation darstellen, und unter diese Kategorie fallen (turing-vollständige) Programmiersprachen, ziehen bei der verständlichen Darstellung von Aspekten nahezu fast immer den Kürzeren. Dies bedeutet aber nicht etwa, dass die MDA den Ansatz verfolgt, Programmiersprachen vollständig durch andere Notationen zu ersetzen. Es kommt vielmehr darauf an zu erkennen, wann es bessere Alternativen zur Lösung eines Problems gibt, also zusätzlich zum Quellcode weitere Sichten auf das behandelte System notwendig sind, um die Komplexität eines Systems aufzubrechen.

Domain-Specific Languages

Auf dem Weg weg von „einer Sprache für alles“ sind *Domain-Specific Languages (DSL)* [DSL] ein weiteres Konzept zur Vereinfachung komplexer Sachverhalte. Unter dem Begriff Domäne wird in diesem Zusammenhang ganz allgemein ein Wissens- bzw. Arbeitsgebiet gefasst. Eine DSL ist demnach eine Sprache, die darauf spezialisiert ist, eine genau abgegrenzte Gruppe von Aspekten darzustellen und damit für ein besseres Verständnis und eine bessere Ausdrucksmöglichkeit von genau abgegrenzten Sachverhalten zu sorgen. Dies erreicht sie durch die Bereitstellung entsprechender Abstraktionen und Notationen für die Konzepte dieser Domäne.

Durch die Verwendung grafischer genau auf die jeweilige Domäne zugeschnittenen DSLs kann die Verständlichkeit von Spezifikationen oft nochmals in erheblichem Maße gesteigert werden. So ist beispielsweise die Darstellung eines Datenbankschemas als Entity Relationship Modell (grafische DSL) bereits intuitiv um einiges verständlicher, als die textuelle Beschreibung in der Data Definition Language (DDL), die eine textuelle DSL der Domäne darstellt.

Man sieht dies in Abb. 3.5 verdeutlicht. Dargestellt ist ein Datenbankschema bestehend aus zwei Tabellen `Person` und `Adresse`. `Person` sind hierbei über eine Fremdschlüsselbeziehung einer oder mehreren `Adressen` zugeordnet. Da beide Darstellungen denselben Sachverhalt modellieren, heißen sie *äquivalent*.

Sprachen wie die DDL bzw. SQL, also solche die das Etikett „DSL“ tragen, sind in der Regel nicht turing-vollständig, beziehen sie doch gerade ihren Nutzen daraus spezifisch und nicht universell zu sein. Die Frage allerdings, ob DSLs überhaupt turing-vollständig sein

dürfen endet oftmals im sprichwörtlichen Streit um des Kaisers Bart, an dem wir uns an dieser Stelle nicht beteiligen wollen.

```
CREATE TABLE Person  (
    PID       number (4),
    Nachname  varchar(30),
    Vorname   varchar(30),
    CONSTRAINT PID_pk PRIMARY KEY(PID),
    CONSTRAINT AID_fk FOREIGN KEY(AID)
                      REFERENCES Adresse(AID)
)

CREATE TABLE Adresse  (
    AID         number (4),
    Strasse     varchar(30),
    Hausnummer  varchar(10),
    Stadt       varchar(80),
    CONSTRAINT AID_pk PRIMARY KEY(AID)
)
```

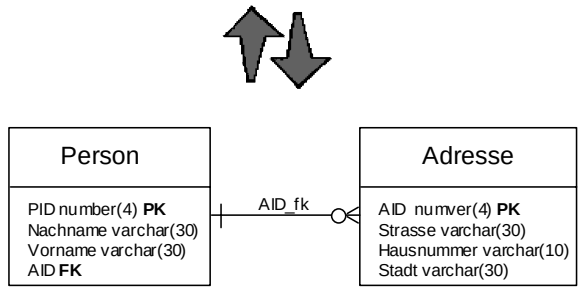


Abb. 3.5 Erstellung eines ER Diagramms aus SQL

3.1.3 Kurze Rekapitulation

Modelle helfen dabei, komplexe *Systeme* in den Griff zu bekommen. Durch die Erstellung von Beschreibungen auf unterschiedlichen *Abstraktionsniveaus* können Systeme in *Module* zerlegt und so deren Komplexität beherrschbar gemacht werden. Modelle entstehen durch die Anwendung von *Modellierungssprachen*. Diese Sprachen können rein *textuell*, rein *grafisch* oder eine *hybride* Mischform aus beiden Ansätzen sein. Dabei gilt der Sinnspruch „ein Bild sagt mehr als tausend Worte“ – man sollte aber immer bereit sein anzufügen: „… aber manchmal reicht auch der Text“. Die geeignete Beschreibungsform hängt dabei stark von den Aufgaben ab, die das resultierende Modell erfüllen soll.

Formale Sprachen – und damit auch die Modelle, die mit ihnen beschrieben werden – zeichnen sich dadurch aus, dass ihre *Syntax*

(die Regeln, die festlegen, aus welchen Elementen die Sprache besteht und wie diese kombiniert werden dürfen) und ihre *Semantik* eindeutig definiert ist. Wir wollen diese Eigenschaft als deterministische *Auswertbarkeit* der Sprachen interpretieren. Dies bedeutet, dass zwei Betrachter, die die Regeln zur Anwendung der Modellsprache kennen, zu genau derselben Deutung der dargestellten Informationen kommen müssen. Eine solche Eigenschaft ist unerlässlich, um die maschinelle Auswertbarkeit der Modelle sicherzustellen, aber auch zur Ermöglichung der personenunabhängigen Konservierung des in den Modellen vorhandenen Wissens. Auch wenn die „Erzeuger“ des Modells nicht (mehr) verfügbar sind, muss die Semantik klar sein.

Um zu entscheiden welche Sprache(n) zur Modellierung eines Systems und seiner unterschiedlichen *Aspekte* verwendet werden soll(en), ist die Fähigkeit der Sprachen, allein oder in der Summe eine ausreichende Menge Semantik transportieren und diese auf verständliche Weise darstellen zu können, entscheidend. Eine Sprache, die sich auf die Beschreibung eines genau abgegrenzten (Fach-)Gebiets beschränkt, gehört in die Klasse der *Domain-Specific Languages (Domänen-Spezifische Sprachen)* und besitzt durch ihre relative Einfachheit gegenüber den „Alleskönnern“ unter den Beschreibungssprachen den Vorteil, dass die wesentlichen Aspekte vom Betrachter schneller aufgenommen werden können. Zur Abdeckung aller Aspekte eines Systems wird nicht selten mehr als eine DSL zur Beschreibung eingesetzt.

Diesen Abschnitt abschließend wollen wir uns noch einmal die Anforderungen anschauen, die wir im Laufe der Betrachtungen an Modelle gestellt haben. Eine sinnvolle Verwendung von Modellen kann im Allgemeinen nur dann erfolgen, wenn sie:

- korrekt sind (also keine syntaktischen oder semantischen Fehler enthalten),
- den zu modellierenden Sachverhalt in einer verständlichen Art und Weise darstellen,
- die einzelnen Sichten auf das System konsistent sind und keine Widersprüche zwischen den einzelnen Darstellungen auftreten,
- sie formalen Ansprüchen genügen,
- vollständig sind, sie also alle Informationen zur Durchführung der Aufgabe, für die sie erstellt wurden, enthalten.

3.2 Unified Modeling Language (UML)

In diesem Buch wollen wir der Empfehlung der OMG folgen und uns im Folgenden auf die Erstellung von Modellen mittels der *UML* (aktuell in Version 2.x) beschränken. An dieser Stelle wird ein Gesamtüberblick über dieses mächtige Werkzeug geboten. Für Details zu den einzelnen Diagrammarten verweisen wir auf die im Anhang A befindliche UML-Schnellreferenz.

UML ist eine Modellierungssprache, die die Spezifikation von Systemen unabhängig von Fach- oder Spezialgebieten ermöglicht. Dazu definiert sie sowohl Notationselemente und Beschreibungen zu deren korrekter Anwendung als auch Erweiterungsmechanismen, um eigene Diagrammarten bzw. Modifikationen von Diagrammen auf Basis dieser Standardkonstrukte erstellen zu können.

Weiterhin können auch völlig neue Modellierungssprachen ohne Vorlage, also quasi auf der grünen Wiese, erstellt werden. UML ist weder Methode noch Prozessmodell und will dies auch nicht sein. Das bedeutet, dass keinerlei Anleitungen mitgeliefert werden, welche Diagrammart wann im Entwicklungsprozess zu benutzen ist. Dies bedeutet zum einen die Freiheit auszuwählen, führt zum anderen jedoch gerade bei „neuen“ Nutzern dieses mächtigen Handwerkzeugs zu Konfusion, welches Diagramm denn nun für den gewünschten Zweck geeignet sei.

Hier sind Erfahrung und Ausprobieren gefragt. Im Rahmen der MDA soll Ihnen das vorliegende Buch die Entscheidungen erleichtern, indem verschiedene Möglichkeiten des Einsatzes vorgestellt werden, die aus realen Projekten stammen oder gar anerkannte *Best-Practices* sind und so ihre Tauglichkeit bewiesen haben.

Es folgt ein kurzer Überblick über die Entwicklung der UML von ihren Anfängen bis heute. Danach wollen wir näher auf die Frage eingehen, warum gerade diese Sprache gewählt wurde, um den MDA-Ansatz zu realisieren. In späteren Teilen dieses Kapitels werden dann die Struktur der UML sowie auch die wichtigsten Prinzipien „unter der Haube“ genauer unter die Lupe genommen.

3.2.1 Historisches

Die Geschichte der UML ist eine sehr dynamische und beginnt zur Zeit der so genannten *Methoden Kriege (engl. method wars)* in den 90er-Jahren.

Zu diesem Zeitpunkt sprossen mehr als 50 Software-Entwicklungsmethoden, die alle ihre eigenen Modellierungssprachen mitbrachten, aus dem Boden und machten eine unterstützende

Zusammenarbeit zwischen Anhängern verschiedener Methoden nahezu unmöglich. Nach und nach kristallisierten sich einige wenige Ansätze heraus, die mehr als nur ein paar Bewunderer unter sich vereinen konnten, zum *de facto* Standard und adaptierten nach und nach die besten Modelltypen aus den Methoden, die auf der Strecke geblieben waren.

Die glücklichen Überlebenden waren (vereinfacht):

- *OOSE [JCJO92] (Object-Oriented Software Engineering)* von Ivar Jacobson,
- *OMT [RBP+91] (Object Modeling Technique)* von James Rumbaugh sowie das
- *OOD [Boo93] (Object-Oriented Design)* von Grady Booch.

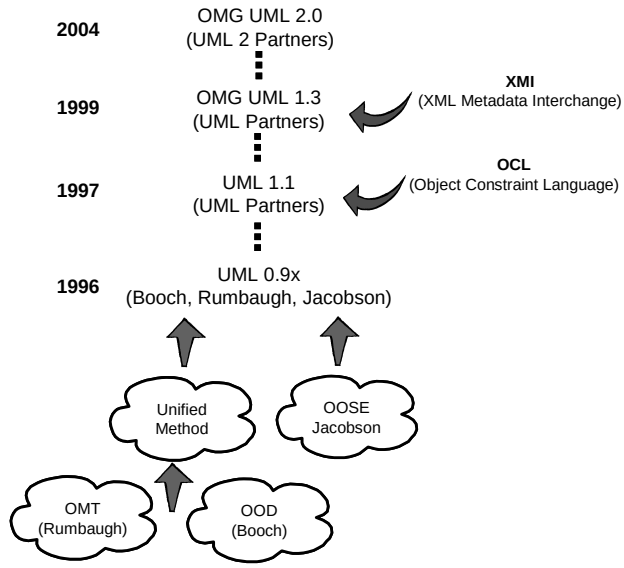


Abb. 3.6 Evolution der UML

Auch diese brachten nicht nur ihre eigenen Prozessmodelle mit sich, sondern hatten wiederum alle ihre eigenen Notationssprachen.

Heftige Debatten über die beste aller Methoden entbrannten, obwohl sich die Ansätze im Kern (objektorientierte Software-Entwicklung als Basis) nicht wesentlich unterschieden. Dies entdeckten alsbald auch die ursprünglichen Erfinder, die im IT-Slang bereits zu „Methoden-Gurus“ aufgestiegen waren und sie beschlossen fortan zusammenzuarbeiten (Abb. 3.6).

Booch und Rumbaugh, die beide bei der Rational Corporation arbeiteten, machten den Anfang und stellten 1995 die erste Version der *„Unified Method“* vor. OOSE wurde derselbigen einverleibt, als Rational die schwedische Firma Objectory übernahm und so auch Ivar Jacobson ins Boot geholt wurde. Unter dem Namen „Die drei Amigos“ widmeten sie sich nun der weiteren Vereinheitlichung ihrer Vorgehensweisen in eine Einzige und stellten 1996 die Früchte ihrer Arbeit unter dem Titel *„Unified Modeling Language“* der gespannten Öffentlichkeit vor.

Nicht verschwiegen werden soll, dass sie sich bei Ihren Bemühungen großzügig bei anderen Ansätzen bedienten: z. B. bei der Entwicklung der *Statechart Diagramme*, die eine Variante des Harel *Zustandsdiagramms* darstellen [Har87].

Der Erfolg ihrer Bemühungen war und ist nie unumstritten gewesen. Die größten Kritiker nennen sie u. a. *„Unwanted Modeling Language“*, was nicht gerade Begeisterung für die Bemühungen der OMG im Feld der UML und deren Propagierung der MDA zum Ausdruck bringt (Näheres hierzu unter [Fow04b]).

Der weitere Werdegang im Schnelldurchlauf

1997 steuerte Rational die in ihren Mauern entwickelte *Object Constraint Language (OCL)* bei, eine textuelle, auf mathematischen Prinzipien aufbauende Sprache, die die Formulierung von Bedingungen erlaubt, die bei Bedarf an Notationselemente angehängt werden können und so den Gedanken des *Design by Contract* [DBC] unterstützt. Bis zur aktuellen UML-Version 2 fristete diese allerdings mehr ein Schattendasein und kommt erst jetzt, gerade im Kontext der MDA, zur vollen Bedeutung.

1998 übergab Rational die Rechte an die *Object Management Group*, die 1999 mit der Version 1.3 die erste „freie“ Version der UML vorlegte. Zur Premiere wurde die Spezifikation zum *XML Metadata Interchange (XMI)* hinzugefügt, die das XML [XML] basierte Austauschformat für (Objekt-) Modelle, unter anderem auch für UML-Modelle darstellt. Obwohl noch nicht endgültig verabschiedet, liegt zurzeit (2006) die Version 2.1 vor, die wesentliche Änderungen und Umstrukturierungen der UML mit sich bringt.

3.2.2 UML – Die Sprache der Model-Driven Architecture

Nach diesem kurzen Abriss des UML-Werdegangs wird es Zeit, sich mit der Frage zu beschäftigen, welche Eigenschaften die UML zur Sprache der Wahl innerhalb der MDA gemacht haben.

Zuallererst werden sowohl die UML als auch die MDA von der OMG unterstützt und weiter entwickelt. Damit bietet sich die UML in natürlicher Weise als Sprache der MDA an. Neben diesem unbestreitbaren initialen Vorteil bietet die UML (und hierbei vor allem die Versionen ab 2.0) jedoch tatsächlich viele Eigenschaften, die ideal zur Erfüllung der an eine gute Modellierungssprache gestellten Anforderungen passen:

MOF

Da ist zunächst einmal das mit Mitteln der *Meta Object Facility (MOF)* formal definierte Metamodell der UML. Auf Metamodelle und die MOF gehen wir in Abschnitt 3.3 im Detail ein. Die Vorteile, die dieses Vorgehen bietet, können wir aber bereits an dieser Stelle kurz skizzieren:

Vereinfacht gesagt trennt der Aufbau der UML auf einem formal definierten Metamodell die Notation eines Modells von der darstellungsunabhängigen Struktur und der Bedeutung (vgl. Ausführungen in Abschnitt 3.1.2 zum Thema Abstrakte Syntax vs. Konkrete Syntax). Beispielsweise ist das (abstrakte) Konzept der Vererbung in der MOF völlig losgelöst von deren konkreter Darstellung in UML-Diagrammen (im Klassendiagramm der Vererbungspfeil) definiert. Da wir die verständliche Darstellung von Systemaspekten als grundlegend für eine sinnvolle Verwendung von Modellen erkannt haben, wird hier der Wert dieser klaren Trennung der visuellen Darstellung von dem eigentlichen Konzept und dessen Semantik deutlich: Ohne die Semantik verändern zu müssen, kann die Notation verwendet werden, die den gewünschten Aspekt der Zielgruppe am verständlichsten darstellt.

UML-Profile

Diese Eigenschaft knüpft nahezu nahtlos an eine weitere prominente Fähigkeit der UML an: die der *Erweiterbarkeit*. Mit der Möglichkeit, auf Basis der UML eigene Erweiterungen und Spezialisierungen der Sprache für besondere (Fach-)Gebiete in Form einer DSL zu erstellen, erfüllt sie die Anforderungen, die wir schon bei der Betrachtung zu formalen Modellen (3.1.2) als grundlegend erkannt haben. Der Mechanismus, der hier greift, ist der der *UML-Profile*, die in 3.3.4 im Detail beschrieben werden. Es können aber auch Sprachen von Grund auf – sozusagen von der grünen Wiese aus – erstellt und spezifiziert werden. Dies geschieht dann mit den Werkzeugen der MOF, mit deren Hilfe komplett eigene Metamodelle erstellt werden können. Weiterhin sind Anwendungen, wegen des formalen Charakters der Sprache, in der Lage automatisch Quellcode oder weitere Modelle, die das System aus weiter differenzierte-

ren Blickwinkeln darstellen, bzw. das Abstraktionsniveau verändern, aus UML-Diagrammen zu erzeugen.

Zusätzliche Abstraktion durch UML

Die Möglichkeit beliebige DSLs in UML zu spezifizieren macht dem Anwender das mächtige Werkzeug der Abstraktion verfügbar. Die Behauptung aus dem vorhergegangenen Abschnitt, diese Abstraktion von Systemen helfe beim Verständnis komplexer Systeme und UML-Diagramme seien trotz ihres formalen Charakters als Modellierungssprache oft verständlicher als reiner Quellcode, soll nun endlich durch ein konkretes Beispiel motiviert werden.

Die Vorschaltung von Modellen vor den eigentlichen Quellcode ermöglicht es, die semantische Beschreibung der Modelle von der Spezifikation der daraus zu generierenden Implementierungen zu trennen. Wir werden später unter dem Begriff der *Plattformunabhängigkeit* auf diesen Sachverhalt zurückkommen.

Beispielsweise ist es möglich, die Verteilung von Komponenten auf verschiedene Rechnerknoten sowie deren Abhängigkeiten untereinander zu modellieren, ohne sich festlegen zu müssen:

- welches Komponentenmodell oder
- welche Middleware

dabei zur Anwendung kommen sollen. Selbst das verwendete Protokoll in der Kommunikation zwischen den Komponenten kann völlig außer Acht gelassen werden, wobei trotzdem eine eindeutige Modellierung ihres Verhaltens untereinander möglich ist.

Versuche, dies z. B. mit Java Quellcode zu bewerkstelligen, sind zwar möglich (Java selbst ist per se plattform-unabhängig) und die Benutzung abstrahierender Frameworks ist hier eine weit verbreitete Technik, um weitere Details zu kapseln und so für höhere Schichten transparent zu machen). Aber unweigerlich wird die Komplexität des Quellcodes vergrößert und macht es in späteren Phasen schwieriger, neu zum Projekt hinzukommende Entwickler einzuarbeiten. Selbst Entwickler, die ständig mit dem Projekt betraut sind, bekommen im Laufe der Zeit Probleme, den Überblick über alle Aspekte der Applikation zu behalten. Die visuelle Darstellung mittels geeigneter UML-Konstrukte jedoch lässt sich in allen Phasen eines Entwicklungsprozesses leicht vermitteln.

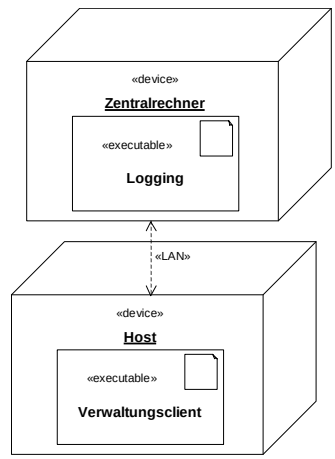
Ein erstes Beispiel

Schauen wir uns ein Beispiel (modelliert mit Konstrukten der UML 2) an, das zwar noch keine speziellen MDA-Merkmale besitzt, jedoch ausreicht, um das eben Behauptete nachzuvollziehen:

Abb. 3.7 zeigt die Verteilung zweier ausführbarer *Einheiten* (`executables`) auf unterschiedlichen *Geräten* (`devices`), die über ein lokales Netzwerk (`LAN`) die Möglichkeit besitzen miteinander zu kommunizieren.

Zudem nennt das Modell auch die Namen der beteiligten Komponenten, nämlich `Zentralrechner` und `Verwaltungsclient` auf Seiten der Geräte (was vermuten lässt, dass es sich um eine Art „Computer“ handeln muss) und `Logging` sowie `Verwaltungsclient` als ausführbare Einheiten.

Abb. 3.7 Komponenten im Verteilungsdiagramm



Dass es sich bei den ausführbaren Einheiten tatsächlich um Komponenten im Sinne der komponentenbasierten Software-Entwicklung handelt (bzw. um eine ausführbare Instanz von solchen), zeigt sich im Komponentendiagramm in Abb. 3.8. Die eingebetteten (assoziierten) Use Cases verraten zudem die Existenz des Anwendungsfalles `Überwache Vorgang` in Komponente `Logging` sowie `Datei speichern` in `Verwaltungsclient`.

Dass die prinzipielle Möglichkeit der beiden zur Kommunikation über das lokale Netzwerk besteht, wissen wir bereits aus Abb. 3.7. Die Gewissheit, dass tatsächlich davon Gebrauch gemacht wird, liefert uns ihre Abhängigkeitsbeziehung im Komponentendiagramm.

`Verwaltungsclient` ist, sowohl auf Komponenten-, als auch auf Ausführungsebene, abhängig von `Logging`, wobei die Abhängigkeit zur Laufzeit lediglich auf dem Wege der Netzwerkkommunikation erfüllt werden kann, was in ihrer Verteilung auf unter-

schiedlichen Knoten begründet liegt und nur die Verbindung über ein LAN besteht.

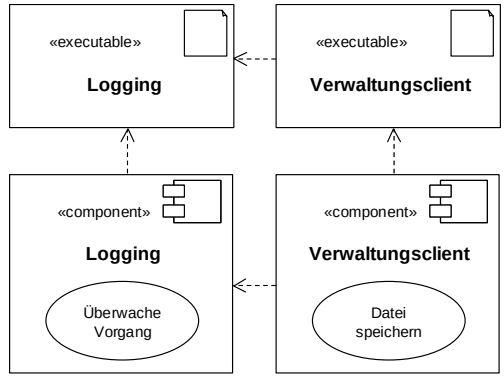


Abb. 3.8 Komponenten-diagramm des Beispiels

Genaueres über die Abhängigkeit oder Kooperation erfahren wir in der nächsten Abbildung (Abb. 3.9).

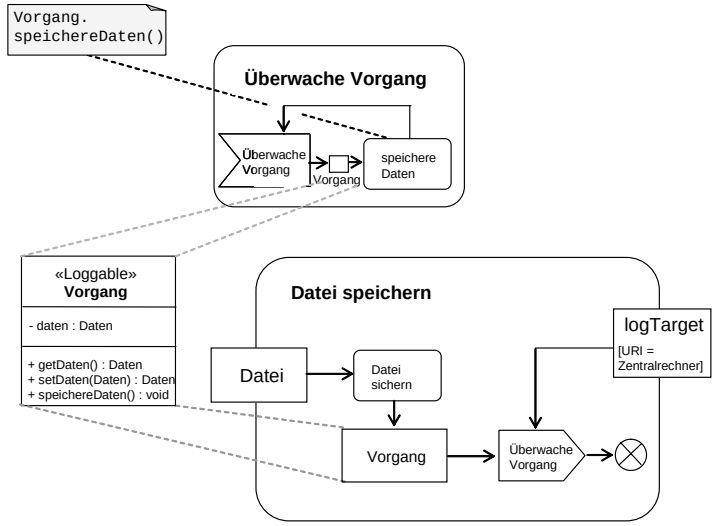


Abb. 3.9 Aktivitäts- und Klassendia-gramm

Will ein Benutzer der Komponente `Verwaltungsclient` eine Datei sichern (Use Case `Datei speichern`), wird ein nicht näher definiertes Objekt vom Typ `Datei` der Aktion `Datei sichern` übergeben, die eine Instanz der Klasse `Vorgang` erzeugt (die grauen Striche deuten deren Spezifikation in einem hier nicht komplett gezeigten Klassendiagramm an), in der Näheres zu der durchgeführten Handlung (wie zum Beispiel: Uhrzeit, Benutzer, …)

festgehalten wird und dann der Aktivität `Überwache Vorgang` ein Aktivierungssignal sendet, die sich in der Komponente `Logging` im Use Case gleichen Namens befindet.

Ein Objekt namens `logTarget` bestimmt dabei die Position des Empfängers über einen Unified-Ressource-Identifier (URI), welcher im vorliegenden Fall den Wert `Zentralrechner` besitzt.

`Überwache Vorgang` führt dann die eigentliche Protokollierung aus, indem sie die Methode `speichereDaten()` von `Vorgang` aufruft, und danach auf neue Benachrichtigungen wartet.

Wir sehen an diesem Beispiel die Fähigkeit der UML strukturelle aber auch dynamische Aspekte eines Systems verständlich darzustellen (vergleichen Sie die Diagramme mal mit der textuellen Beschreibung im obigen Abschnitt!). Dass das Ganze auch formal und detailliert genug geschehen kann, um die Generierung von Anwendungen (zumindest in Teilen) zu realisieren und eine Wiederverwendung von Anwendungswissen ermöglicht, werden wir im weiteren Verlauf des Buches, der sich mit der praktischen Umsetzung der theoretischen Erkenntnisse dieses ersten Teils beschäftigt, versuchen zu verdeutlichen.

Zuvor jedoch soll der nun folgende Überblick über den Aufbau der UML 2 und ihrer Diagrammtypen der Information sowie der Vorbereitung auf diesen praktischen Teil dienen.

3.2.3 UML-Spezifikationen

Besucht man die Seiten der Object Management Group, um sich von dort die UML-Spezifikation [UML] zu besorgen, muss man zunächst erfahren, dass die eine Spezifikation gar nicht existiert, sondern die UML vielmehr aus einer ganzen Sammlung von Einzelspezifikationen besteht, denen man sich widmen muss. Aufgrund der Menge Holz, die nötig wäre, wollte man alle Spezifikationen in ausgedruckter Form sein Eigen nennen, heißt die Gesamtmenge dieser Spezifikationen auch *UML-Stack* (dt. in etwa Stapel). Wenn wir im Folgenden abkürzend von der UML sprechen, meinen wir genau diesen Stapel und den in der Version 2.

UML-Infrastructure

Nichtsdestotrotz können fünf Kernelemente der UML ausgemacht werden, die in Abb. 3.10 dargestellt sind. Architekturelle Grundlage des Gesamtgebäudes UML ist die *UML-Infrastructure*. Da alle Metamodelle der UML unter Nutzung der MOF definiert werden und Instanzen des dort definierten „Urmodells“ sind und diese die

Elemente der Infrastructure lediglich wiederverwendet und ergänzt, bildet die Infrastructure die Basis aller UML-Metamodelle. Sie definiert die abstrakte Syntax der grundlegenden Sprachkonstrukte sowie die Konzepte, auf denen die konkrete Notation der UML (Klassen, Assoziationen usw.) aufbaut und fasst diese in der so genannten *InfrastructureLibrary* zusammen.

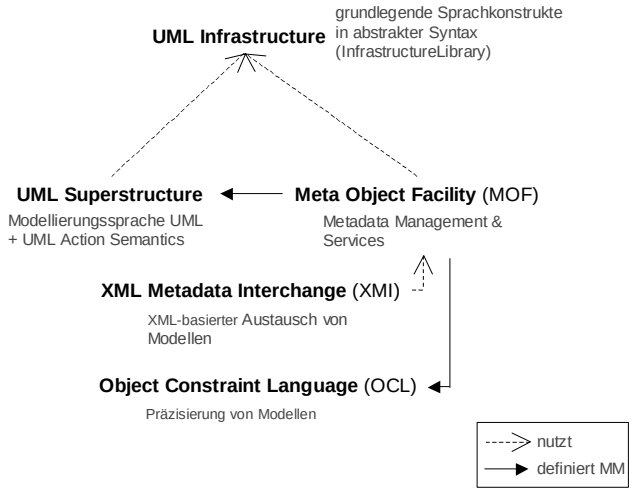


Abb. 3.10 Der Aufbau des UML 2.x Stacks

Die Möglichkeit der Erweiterung der in der Infrastructure definierten Elemente durch eigene Konstrukte wird durch den Mechanismus der *Profile* realisiert, der an dieser Stelle der Spezifikation erstmals definiert ist. Näher erläutert wird diese Möglichkeit in Abschnitt 3.3.4.

MOF & XMI

Zur Erstellung von Metamodellen bietet die UML die *Meta Object Facility (MOF)*. Sie importiert die in der Infrastructure definierte abstrakte Syntax und erweitert sie um Dienste, die im Bereich des Metamodel Managements angesiedelt sind.

Zur Erzeugung von Metamodellen finden hier dieselben Elemente Verwendung, die innerhalb der UML auch für die Klassenmodellierung genutzt werden. Für den Austausch der Modelle (zwischen Modellierungswerkzeugen oder als Eingabe für Generatoren) steht ein Format zur Verfügung, das in der *XML Metadata Interchange (XMI)* Spezifikation definiert ist. Auf die MOF wird in Abschnitt 3.3.2 näher eingegangen.

Superstructure

Die *UML-Superstructure* ist der Teil der UML mit der wir, vielleicht ohne es zu wissen, bereits am vertrautesten sind. Sie definiert die bekannten Diagrammarten wie beispielsweise *Klassen-* oder *Aktivitätsdiagramme*, die dem Endbenutzer zur Modellierung mit UML zur Verfügung stehen.

Hier wird sowohl festgelegt welche Elemente (grafische Symbole) wir zur visuellen Modellierung benutzen dürfen, als auch welchen Kombinationsregeln wir uns unterwerfen müssen (abstrakte Syntax), um unser Modell UML konform nennen zu dürfen. Die Superstructure stellt damit das Metamodell der UML-Diagramme dar.

Action Semantics

Die Action Semantics der UML sind ebenfalls Bestandteil der Superstructure. Durch sie wird die präzise Modellierung dynamischen Verhaltens ermöglicht. Näheres zur Action Semantics findet sich in Abschnitt 3.5).

Erweiterungen der Infrastructure Basiskonzepte werden durch eine Kombination von OCL und Prosa (natürlichsprachliche Erklärung) beschrieben. *OCL*, die *Object Constraint Language*, ist eine formale, textbasierte Sprache zur Präzisierung von Modellen, die auf mathematische Konzepte aufbaut. Dabei wurde Wert darauf gelegt, diese möglichst intuitiv verständlich zu machen. Nähere Informationen zu diesem wichtigen Bestandteil des UML-Stacks finden Sie in Abschnitt 3.6.

Änderungen UML 1.x – UML 2

Sieht man von den weit reichenden Veränderungen innerhalb der einzelnen Diagramme, die in den entsprechenden Abschnitten der mitgelieferten UML-Schnellreferenz (Anhang A) teilweise erläutert werden, einmal ab, ist die grundlegendste Veränderung innerhalb der UML in ihrer Struktur der einheitliche Aufbau aller Spezifikationen auf Konzepten der UML-Infrastructure. Da in der aktuellen Version alle architekturellen Basiskonzepte der UML innerhalb der zur Infrastructure gehörenden *InfrastructureLibrary* zusammengefasst sind und in den Instanziierungen dieser Konstrukte innerhalb der auf ihr aufbauenden Spezifikationen lediglich Erweiterungen der dort definierten Elemente vorkommen, greift die Mächtigkeit der Polymorphie, sodass zum Beispiel alle in der MOF angebotenen Dienste, und damit auch die Möglichkeit des Austausches mittels XMI, auf alle Modelle angewandt werden können.

Classifier

Besondere Rolle spielt in diesem Zusammenhang das abstrakte Element des Typs `Classifier`, das das Konzept der Klasse weiter abstrahiert und von vielen Elementen der UML 2 direkt realisiert wird. Da viele Beziehungen zwischen Elementen der UML 2 schon auf `Classifier`-Ebene spezifiziert sind, ergeben sich eine Vielfalt von Möglichkeiten, die Konstrukte aus unterschiedlichen Diagrammtypen verbinden bzw. die Abhängigkeiten/Zuordnung ganzer Diagramme zueinander visualisierbar machen. Der Gewinn gegenüber der unstrukturierteren 1.x Versionen ist, dass dadurch viele Konzepte nur einmal gelernt und dann konsistent auf alle Elemente, die `Classifier` spezialisieren, angewendet werden können (von diesem Typ erben).

Diagrammarten der UML 2

Prinzipiell stehen dem willigen Anwender dreizehn unterschiedliche Diagrammarten zur Verfügung, die alle spezielle Charakteristiken des modellierten Sachverhalts in den Vordergrund stellen und dafür andere vernachlässigen – kurz: bestimmte Aspekte eines Systems genauer betrachten und dafür andere vernachlässigen.

Während Strukturdiagramme die statischen Aspekte eines Systems unter die Lupe nehmen und so etwas wie den Bauplan eines Systems repräsentieren, nehmen sich Verhaltensdiagramme den dynamischen Teil vor und widmen sich den Änderungen, die die strukturellen Elemente während der Laufzeit des Systems erfahren.

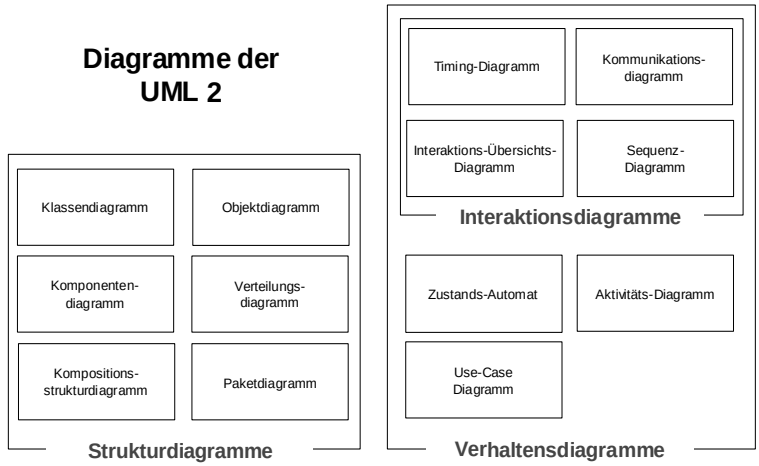


Abb. 3.11 Die 13 Diagrammarten der UML 2

An dieser Stelle wollen wir nicht näher auf die einzelnen Diagramme und deren Möglichkeiten eingehen, dies geschieht im Anhang A (UML-Schnellreferenz).

3.3 Metamodellierung

Wie wir gehört haben, basieren alle Spezifikationen des UML-Stacks auf der MOF, die die Möglichkeit bietet, eigene Modellierungssprachen durch die Angabe eines Metamodells zu definieren. Das Konzept der Metamodelle bzw. der Metamodellierung ist somit wesentliche Grundlage des UML-Gebäudes, aber auch wesentlicher Bestandteil modellgetriebener Ansätze wie der MDA überhaupt, weshalb wir an dieser Stelle die unterliegenden Konzepte und Techniken näher vorstellen wollen. Vor der Beschäftigung mit den Einzelheiten wollen wir noch einmal motivieren, warum wir so etwas wie Metakonstrukte überhaupt brauchen und dann die theoretischen Grundlagen schaffen, die uns das Verstehen vieler darauf aufbauenden Konzepte der MDA und der folgenden Kapitel erleichtern und teilweise erst ermöglichen.

3.3.1 Meta? – Grundlagen

Grundvoraussetzung zur Durchführung der MDA ist die Möglichkeit, Systeme auf unterschiedlichen Abstraktionsebenen beschreiben zu können. Um dies zu erreichen, bedient sie sich der Technik der Modellierung und benutzt zur Darstellung der unterschiedlichen Aspekte des modellierten Systems Domänen-Spezifische Sprachen, um sowohl die Wiederverwendbarkeit, als auch die Verständlichkeit der Modelle zu erhöhen.

Ein Werkzeug zur Definition eigener Sprachen

Schön wäre also ein Werkzeug, um selbst eigene Modellierungssprachen definieren zu können, also ihre Syntax und Semantik in einer Form zu beschreiben, die sie zu formalen Sprachen erhebt und so zum Beispiel als Eingabe von Generatoren nutzbar macht.

Auch eine Sprache ist ein System

Machen wir einen kleinen gedanklichen Abstraktionsschritt und betrachten eine (Modellierungs-)Sprache selbst als System, so ist eine Beschreibung derselben ein Modell dieser Sprache. Beschreibt ein Modell die Syntax eines anderen Modells (welche Notationselemente gibt es und wie dürfen diese kombiniert werden), nennt man ein solches Modell *Metamodell* bzw. die Sprache, die beschreibt wie „Sätze“ innerhalb einer anderen Sprache auszusehen haben, analog *Metasprache*.

*Metamodelle sind **keine** „Modelle von Modellen“?*

Vorsicht: Obwohl griffig formuliert, immer wieder gehört und zunächst anscheinend einleuchtend, sind Metamodelle mitnichten einfach nur „*Modelle von Modellen*“, sondern korrekter (aber auch sehr viel umständlicher): „*Modelle von Modellierungssprachen* von

Modellen“, da ein konkretes Modell (z. B. ein UML-Diagramm) immer nur einen Ausschnitt der erlaubten Notation zeigt, das Metamodell jedoch die Gesamtheit aller erlaubten Konstrukte (Syntax) darstellt. Atmen wir kurz durch und schauen uns den gerade erläuterten Sachverhalt noch einmal in der folgenden Abbildung (Abb. 3.12) an.

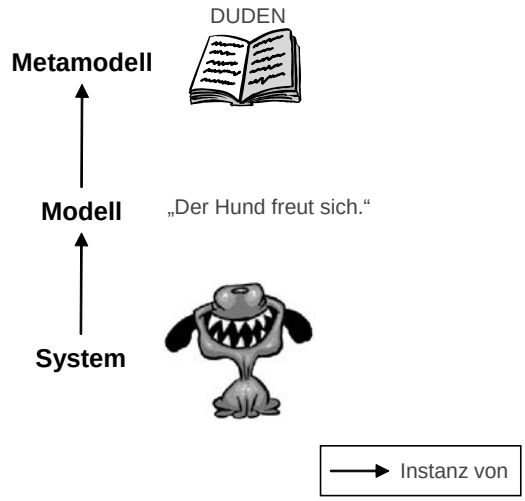


Abb. 3.12 Beziehung Modell – Metamodell

Nehmen wir als Modellierungssprache – die gewählte Notation – die deutsche Sprache und als System einen fröhlichen Hund, so wird der Satz „*Der Hund freut sich.*“ zum *Modell* des Systems, da er es beschreibt. Die Menge aller korrekten Modelle (also alle Sätze, die ohne den Einsatz des Rotstiftes auskommen) ist dabei definiert im DUDEN, der die Regeln zur korrekten Bildung eines Satzes sowie alle deutschen Wörter, die dazu benutzt werden dürfen, beschreibt.

Der DUDEN ist somit *Metamodell* unseres Satzes, unser Satz wiederum eine *Instanz* (konkrete Ausprägung) der im Duden beschriebenen Gesamtheit der korrekten Modelle (Sätze). Synonym zum Begriff Metamodell wird oft von der *abstrakten Syntax* einer Sprache geredet.

Instanziierung und Metalevel

Um den Instanziierungspfeil vom System zum Modell erklären zu können, müssen wir noch ein wenig tiefer in die Welt der Metamodellierung vorstoßen und das Konzept der Metalevel einführen.

Wir haben gesehen, dass das *Präfix Meta* innerhalb der Modellwelt immer in etwa so etwas bedeutet wie „*Dieses Modell, nennen wir es MA, beschreibt, wie andere Modelle aussehen dürfen*“. *MA* ist dann

ein Metamodell. Wenden wir diese Regel rekursiv an, können wir das Metamodel von *MA* mit dem Titel *MetaMetaModell* versehen. Teilen wir die Modelle nach der Anzahl der „Metas“ ein, erhalten wir die *Metalevel* M0 – Mn. Faustregel: `Mx+1` ist Metamodell von `Mx`. Mit größer werdendem `x` steigt das Metalevel, die Beschreibung wird abstrakter.

So weit so gut, aber was können wir uns unter einem MetaMeta…-Modell konkret vorstellen? Antwort bietet die folgende Tabelle (Tabelle 3.1).

Wie bereits erwähnt sind Modelle mit kleinerer Nummer direkte oder indirekte Instanzen von Modellen größerer Nummer. M0 hat dabei insofern eine Sonderstellung, dass es eine Art Anker darstellt und die reale Welt repräsentiert. Noch realer (M-1, …) geht es nicht, nach oben (abstrakter) ist keine Grenze gesetzt. Wie wir sehen werden, sind aber auch hier nur die Level bis M3 von mehr als philosophischem Interesse.

Tabelle 3.1 Metalevel Übersicht

Metalevel	Name	Beispiel
M3	Urmodell	MOF
M2	Metamodelle	UML-Superstructure, OCL
M1	Modelle	Klassendiagramm
M0	Systeme	Anwendung, Hund

Beginnen wir mit der Beschreibung von M1, da wir als Anwender häufig Modelle dieser Ebene erstellen: M1 bezeichnet alle Instanzen von M2 Modellen. Modelle bestehen aus Elementen, die wiederum Instanzen von Elementen des zugehörigen Metamodells sind (hier aus M2). Ein Klassendiagramm, das ein Auto (Teil der realen M0 Welt) mittels der Klasse `Auto` (Element in M1) beschreibt, ist ein M1 Modell. Das Konstrukt Klasse (`Class`) wiederum stammt aus einem Modell der M2 Welt. Bezogen auf unser Klassendiagramm ein Metamodell (M2), das in Form der UML-Superstructure gegeben ist.

Insgesamt definiert die Meta Object Facility (MOF) vier Metalevel M0 – M3, wobei die MOF selbst den dritten Level darstellt, den wir hier als *Urmodell* bezeichnen wollen, da letztendlich alle anderen Modelle auf ihr basieren.

Wie jedoch sieht das Metamodell der MOF selbst aus, die ja auch eine Sprache zur Modellierung von Metamodellen ist, wenn es nur vier Metalevel gibt? Bräuchten wir nicht eigentlich *mindestens* fünf Ebenen? Die Antwort ist „*Ja.*“, wir bräuchten sogar unendlich viele Ebenen, würden wir den Ansatz so uneingeschränkt weiterverfolgen. Die Macher der UML haben sich allerdings entschieden, dieses nicht zu tun und kurzerhand die MOF als selbstbeschreibend definiert. Sie beschreibt sich unter Nutzung von Konstrukten und Mechanismen, die sie selber definiert, wodurch die Anzahl der notwendigen Metalevel auf vier beschränkt bleibt.

Die MOF definiert sich selbst

3.3.2 Meta Object Facility 2 (MOF 2)

Wie bereits im vorigen Abschnitt kurz angesprochen, stellt die *Meta Object Facility 2 (MOF 2)* das Werkzeug der UML zur Erstellung und Erweiterung von Metamodellen dar. Dazu bedient sie sich Konstrukten, die uns bereits aus den anderen Spezifikationen der UML bekannt sind: Elemente von Metamodellen werden als Klassen mit Attributen, Operationen und Assoziationen untereinander beschrieben. Instanziiert entstehen aus ihnen Modelle, die dann eine Stufe tiefer auf der Metalevel-Ebene angesiedelt sind.

Im Folgenden soll der grundlegende Aufbau der MOF sowie einige dort definierte repräsentative Elemente vorgestellt werden. Da es sich hierbei um ein zwar interessantes, jedoch sehr komplexes Thema handelt, dessen ausführliche Behandlung den Rahmen dieses Buches sprengen würde, müssen ein kurzer Überblick und ein Hinweis auf die Originalspezifikation [MOF2] für Interessierte genügen.

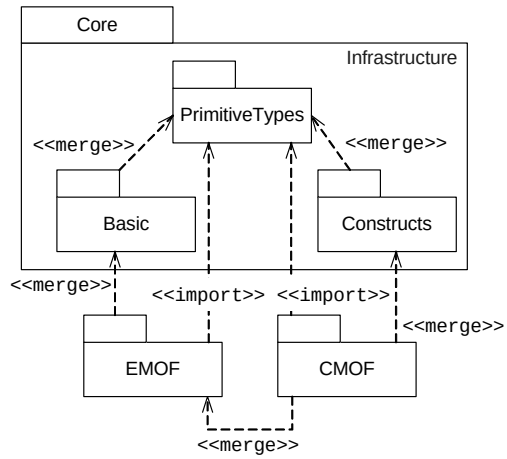
Werdegang und Grundlagen

Der Kerngedanke bei der Konzeption der MOF 2 als Nachfolger der Vorgängerversion 1.4 war der Versuch, aus den Erfahrungen der Vergangenheit zu lernen und eine möglichst vollständige Wiederverwendung bereits bestehender Konstrukte aus anderen UML 2 Spezifikationen zu erreichen. So sollte ein Auseinanderdriften der Spezifikationen, wie es in den Vorgängerversionen der UML-Spezifikationen zum Problem geworden war (z. B. diffizile aber wichtige Unterschiede zwischen UML und MOF), möglichst verhindert werden.

Ergebnis der Bemühungen ist eine Architektur der UML-Spezifikationen, in der die schon betrachtete UML-Infrastructure die Basiselemente sowohl der UML-Superstructure als auch der MOF 2

bereitstellt. Dieser Ansatz des Aufbauens auf einem gemeinsamen Kern soll auch in der Zukunft weiter verfolgt werden. Die OMG spricht von diesem Unterfangen als eine Art „komponentenbasierter Modellierung“ der Metamodelle. Zur Vereinfachung der bestehenden Modelle wurden zudem Techniken eingesetzt, die aus der Domäne der OOP stammen und einem Refactoring der Metamodelle auf Klassen- und Paketebene entsprechen.

Abb. 3.13 Aufbau der MOF 2 – Übersicht



Werfen wir nun einen Blick auf den Aufbau der MOF. Die MOF 2 besteht in der Basis aus zwei Hauptbestandteilen (Abb. 3.13).

Als Grundlage der MOF 2 dient das `Core` Paket der UML-Infrastructure, in der die grundlegenden Bauelemente für Metamodelle wie Klasse, Attribut, Assoziation usw. definiert sind und hier lediglich wiederverwendet bzw. teilweise leicht angepasst werden. Die MOF 2 selbst gliedert sich noch einmal in zwei Hauptbestandteile, die auch die zwei durch die OMG definierten *Konformitätslevel* darstellen:

- dem *Essential MOF Model (EMOF)*, einem Paket, das eine Basisfunktionalität der MOF enthält und
- dem *Complete MOF Model (CMOF)*, das die gesamte Funktionalität der Meta Object Facility umfasst und in der alle anderen Metamodelle des UML-Stacks (wie die UML-Superstructure und selbst die EMOF) definiert sind.

Um sich MOF 2 konform nennen zu dürfen, muss ein Modellierungstool mindestens die in der EMOF definierten Konzepte umsetzen, das Siegel *„100% MOF 2“* erhält es erst, wenn auch alle Elemente der CMOF umgesetzt werden.

Essential MOF Model – EMOF

EMOF ist eine Untermenge der MOF, die die Erstellung einfacher Metamodelle mit Konstrukten ähnlich denen von Programmiersprachen sowie Services zur Manipulation der Modellelemente zur „Laufzeit“ des Modells zur Verfügung stellt. Abb. 3.14 zeigt den architekturellen Aufbau der EMOF.

Wie bereits in Abb. 3.13 dargestellt, enthält EMOF alle Elemente des `Core::Basic` Pakets aus der UML-Infrastructure (wie `Class`, `Operation`, `Package`, …) und erweitert diese um einige Zusatzdienste, die wir uns in aller Kürze anschauen wollen:

`Identifiers` erweitert die MOF um die Möglichkeit, jedem Modellelement eine eindeutige ID im Rahmen eines Kontextes (in der Spezifikation *Extent* genannt) zuzuweisen. Ein Modellelement kann beliebig vielen Extents angehören.

Die Elemente eines Modells (bzw. die Instanzen der Elemente des übergeordneten Metamodells) erhalten damit eine Identität, die mit der *Objektidentität* der OOP vergleichbar ist.

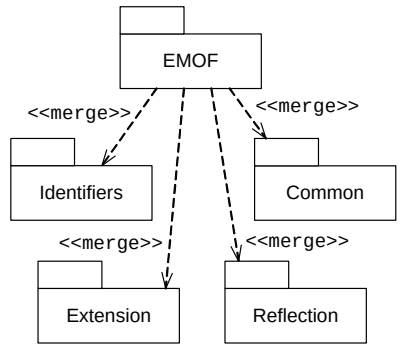


Abb. 3.14 EMOF – Übersicht

Einige Beispiele für die sinnvolle Verwendung dieser Identität:

EMOF:: Identifiers

- Bei der Durchführung von Simulationen, die auf Modellen durchgeführt werden (Stichwort *Executable UML* [MB02]), können Ursachen (zum Beispiel Ereignisse/Signale) und deren

Wirkung (z. B. die Ausführung von Verhalten) eindeutig auf bestimmte, auslösende Modellelemente zurückgeführt werden,

- Obwohl sich durch Modelltransformationen Objekte ändern können, ist die Rückverfolgbarkeit der Transformationskette aufgrund der eindeutigen ID zu jedem Zeitpunkt möglich,
- Referenzen (Assoziationen) zwischen Modellelementen können auch bei der Speicherung in mehreren unterschiedlichen Ablageorten leicht wieder aufgelöst und wiederhergestellt werden.
- usw.

EMOF:: Extension

`Extension` definiert das Element `Tag`, das, einem beliebigen anderen Element zugeordnet, zusätzliche Informationen über dieses speichern kann. Die Art der Information ist ein Tupel aus Name und Wert des Tags und ist in seiner Form keiner Beschränkung unterlegt. `Tags` ersetzen das aus den 1.x Versionen der UML bekannte Konzept der *Tagged Values* und dürfen deshalb auch weiterhin so genannt werden.

EMOF:: Reflection

`Reflection` definiert das Element `Element`, das unter anderem auch Superklasse jedes Elementes der UML-Infrastructure ist und damit auch die Basis für alle Elemente der MOF und UML-Superstructure darstellt. `Element` enthält Operationen, die dem Aufrufer Informationen über die Beschaffenheit des zugehörigen Elementes liefern, d. h. Daten aus dessen Metamodell (z. B. seine (Meta)Klasse/Typ, o. Ä.), ohne vorher etwas über das Modellelement wissen zu müssen. Vom Prinzip her ähneln die Fähigkeiten damit den Fähigkeiten reflektiver Programmiersprachen (wie z. B. Java), mit deren `getClass()` Methode beispielsweise zur Laufzeit die Klasse – im Metamodellkontext die Metaklasse – des betrachteten Objektes ermittelt werden kann.

Complete MOF Model – CMOF

Den architekturellen Aufbau der CMOF zeigt Abb. 3.15. *CMOF* übernimmt per `«merge»` die eben beschriebenen Pakete der EMOF sowie per `«import»` alle Elemente von `Core::Constructs` und erweitert weiterhin durch die in `CMOF::Reflection` und `CMOF::Extension` definierten Konstrukte. Diese wiederum ergänzen die bereits in der EMOF beschriebenen Pakete um weitere reflektive Dienste und hoch spezielle Konstrukte, wie beispielsweise ein *Factory* Konstrukt zur Erzeugung von Modellelementen zur Laufzeit, deren ausführliche Beschreibung an dieser Stelle zu sehr in die Tiefe gehen würde.

Für das weitere Verständnis der MDA-Grundlagen sind sie nicht weiter hilfreich, wir verweisen daher die auch an diesem Thema weiter Interessierten auf die originalen Spezifikationstexte in [MOF2].

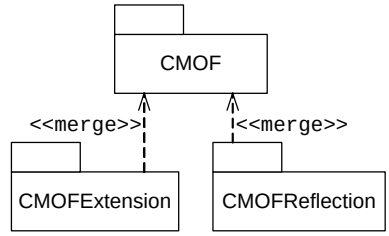


Abb. 3.15 CMOF – Übersicht

Mapping der MOF auf konkrete Datenstrukturen

Stark an die MOF gebunden, obwohl eigenständige Spezifikationen, sind das *Java Metadata Interface (JMI)* [JMI] sowie die der *XML Metadata Interchange (XMI)* [MOF2XMI]. Bei der JMI spricht man vom *„programming-model"* der MOF, da es ein *Mapping* (eine Zuordnung) von Elementen des MOF Metamodells auf Sprachkonstrukte der Programmiersprache Java darstellt.

Man kann sich das im Falle von Java so vorstellen, dass dort Interfaces spezifiziert werden, die die in der MOF definierten Elemente inklusive derer zugehörigen Eigenschaften nachbilden (also z. B. ein `String` für die Identität, eine Operation, die den Typ eines Elementes zurückliefert etc.). Implementierungen dieser Interfaces stellen die geforderten Elemente zur Verfügung und ermöglichen damit die Erstellung eines Objektbaums aus einem Modell, auf dem dann gearbeitet werden kann (Traversierungen mit Code Generierung, Erweiterung, Hinzufügen/Löschen von Modellen, …).

Zur Erstellung des Objektbaumes kann als Quelle eine XMI-Datei herangezogen bzw. nach der Arbeit an einem solchen Baum das entstandene Modell wieder als XMI serialisiert werden [GDB03].

XMI – XML Metadata Interchange

XMI kann ebenfalls als Mapping aufgefasst werden, nämlich als eines der MOF-Strukturen auf das Dateiformat *eXtensible Markup Language (XML)*. Es spezifiziert die Vorschriften zur Serialisierung eines beliebigen MOF-basierten Metamodells und legt die Regeln zur Erstellung von *XML-Schemata* aus diesem fest. Durch die Serialisierung (jede Modell-Klasse wird durch ein XML-Element mit deren Namen repräsentiert, in dessen Deklaration dann die *Properties* (Eigenschaften) dieser Klasse auftauchen) wird der Austausch von Modellen zwischen Anwendungen realisiert. Die Erstellung von Schemen aus Metamodellen wiederum macht eine

Überprüfung der syntaktischen Korrektheit eines Modells bezüglich seines Metamodells mithilfe von XML Parsern möglich. Dabei ist zu beachten, dass nicht die Überprüfung aller Bedingungen, die seitens der MOF an ein Modell gestellt werden können (z. B. OCL-Tags), mittels dieser Technik gelingt, da diese nicht in eine XML Struktur, die vom Parser überprüfbar wäre, übersetzt werden können.

3.3.3 Beispiel für ein Metamodell

Um komplexere *M2*-Modelle verstehen und eventuell sogar selbst erstellen zu können, erläutern wir im Folgenden die Entwicklung eines solchen Modells für eine Modellierungssprache eines einfachen Modelltyps beispielhaft.

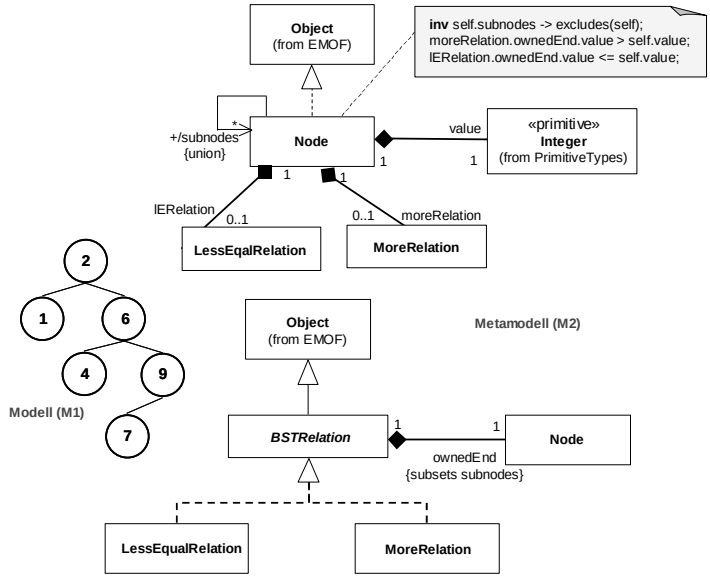


Abb. 3.16 Metamodell eines Binary Search Trees

Als Modell soll die Sprache zur Erzeugung von *Binary Search Trees (BST)* dienen. Binäre Suchbäume besitzen zum einen typische Baumeigenschaften (ein Wurzelknoten, höchstens zwei Kinder pro Knoten, keine Zyklenbildung) sowie zum anderen die Fähigkeit der Knoten einen Wert zu speichern, in diesem Falle eine natürliche Zahl, wobei Werte eines Knotens und die seiner Kinder dabei in einem speziellen Verhältnis zueinander stehen müssen. Wir definieren, dass alle „linken Kinder“ eines Knotens einen Wert, der kleiner gleich, und alle „rechten Kinder“ einen größeren Wert als der Elternknoten besitzen müssen. Der Vorteil dieser Festlegung ergibt sich bei der Suche nach einem Knoten mit einem bestimmten Wert,

da bei jedem Vergleich ein ganzer Zweig des Restbaumes von der Suche ausgeschlossen werden kann.

Auf der linken Seite der Abbildung (Abb. 3.16) sehen wir ein Modell eines BSTs auf Ebene M1. Zur Beschreibung der Sprache, die die Darstellung eines beliebigen Baumes auf M1 Ebene ermöglicht, also ein Metamodell von BSTs, brauchen wir ein M2 Modell, das wir auf der rechten Seite der Abbildung wiederfinden.

Konkret werden folgende Metaelemente neu definiert:

- `Node`, im oberen Teil, und
- `BSTRelation`, `LessEqualRelation` sowie `MoreRelation` im unteren Teil der Abbildung.

Beginnen wir mit der Definition von `Node`: `Node` realisiert die in der EMOF enthaltene Metaklasse `Object` und ist instanziierbar, was bedeutet, dass auch Modelle, die auf unser Metamodell aufbauen, diese Klasse als Notationselement nutzen dürfen. Wir sehen weiterhin, dass jede `Node` mandatorisch genau ein Attribut namens `value` besitzt, das einen Wert des Typs `Integer` speichern kann. Die Klasse `Integer` stammt dabei aus dem Paket `Core::Basic` der MOF.

`Node` kann wahlweise höchstens zwei Elemente, die `BSTRelation` realisieren, enthalten. Sie können über `lERelation` bzw. `moreRelation` angesprochen werden. Weiterhin existiert ein abgeleitetes Attribut `subnodes`, das eine Menge von `Nodes` Elementen darstellt. Woher sich diese Menge ableitet, werden wir später sehen. `BSTRelation` ist eine abstrakte Klasse und muss erst realisiert werden, bevor es als Element in einem BST Modell der Ebene 1 auftauchen darf. Diese Realisierung sehen wir im zweiten, unteren Teil unseres Metamodells.

Auch `BSTRelation` erbt von `Object` und unterhält pflichtmäßig genau ein Attribut vom Typ `Node`, das sich `ownedEnd` nennt. Hier sehen wir auch, wie die Menge `subnodes` in `Node` zustande kommt, es ist die Menge aller `Nodes`, die über `ownedEnd` an einer `BSTRelation` hängen, sprich: die Elemente im Unterbaum eines Knotens. `LessEqualRelation` sowie `MoreRelation` realisieren `BSTRelation`.

Nun zum „Kleingedruckten“ unseres Modells, das von entscheidender Bedeutung ist. Die oben genannten Bedingungen für die Integrität eines binären Suchbaumes werden durch eine *Invariante* (`inv`)

definiert, die alle Elemente vom Type `Node` erfüllen müssen und die mittels *OCL* beschrieben ist. Wie oft üblich ist auch in diesem Diagramm die Invariante in Form eines Kommentars an das zugehörige Element angetragen und enthält folgende Bedingungen (Constraints):

- `self.subnodes->excludes(self)` bedeutet nichts anderes, als dass ein Knoten in keinem seiner Unterbäume enthalten sein darf (Zyklenfreiheit).
- `moreRelation.ownedEnd.value <= self.value` bedeutet, dass Knoten dieses Typs einen kleineren oder gleichen Wert haben dürfen, also „rechts" unter dem Elternknoten liegen.
- `lERelation.ownedEnd.value > self.value` stellt analog den „linken" Teil unter einem Elternknoten dar und muss einen Wert kleiner als dieser besitzen.

Beachten Sie bitte, dass die oben erwähnten Positionsangaben „links" und „rechts" nicht vom Metamodell gefordert werden. Zur Darstellung dieses Sachverhalts müsste Prosa herhalten, die maschinelle Auswertbarkeit des Modells wird aber in keiner Weise davon beeinflusst. Auch die grafische Notation (also `Nodes` als Kreise mit dem Wert in der Mitte, `BSTRelations` als Linien zwischen `Nodes`) muss in Prosa beschrieben werden und ist hier nicht dargestellt.

Neben den explizierten Bedingungen, die in Form von OCL angehangen sind, gelten selbstverständlich auch die, die durch die Verwendung spezieller UML-Konstrukte Gültigkeit für unser Modell erhalten. Erwähnt werden soll hier beispielsweise, dass durch die Kompositionsassoziation von `Nodes` und `BSTRelations` klar geregelt ist, dass bei Löschen eines Elternknotens alle Unterbäume rekursiv mitgelöscht werden (*cascaded delete*).

3.3.4 UML-Profile

Mechanismen, die uns beliebige Veränderungen bis hin zur völligen Neuerstellung von Metamodellen erlauben, wie wir sie im letzten Abschnitt am Beispiel des Metamodells für BSTs gesehen haben, werden *„Erweiterungsmechanismen erster Klasse" (engl. First-Class Extension Mechanisms)* genannt. Oft ist es für viele Zwecke allerdings völlig ausreichend und viel bequemer, bereits definierte Konstrukte wiederzuverwenden bzw. an seine eigenen Bedürfnisse anzupassen.

Die UML-Infrastructure stellt dazu den Mechanismus der UML-Profile zur Verfügung, der die Erweiterung von Metamodellen bzw. deren Metaklassen erlaubt. Im Vordergrund steht dabei tatsächlich die Erweiterung und nicht die Modifizierung bestehender (Meta-)Modelle. So ist es beispielsweise nicht möglich, bestehende Bedingungen, die vom existierenden Metamodell an die Elemente gestellt werden (*„Jedes Element X muss mindestens eine Assoziation zu einem Element Y haben.“* o. Ä.), zu streichen. Lediglich das Hinzufügen in der Sprache des *Design by Contract* [DBC] die *Verschärfung* des Vertrages zwischen Modellelementen ist möglich.

Profile

Mit dem Begriff *Profil* bezeichnet man nun eine benamste Menge solcher Änderungen an einem bestehenden Metamodell. Die Art der erlaubten Änderungen ist dabei durch die UML-Infrastructure genau vorgegeben. Die drei Mechanismen, die hier eine Rolle spielen, sind die der *Stereotypen und Bedingungen (constraints)* sowie das der *TaggedValues*, die bereits im Abschnitt über die EMOF kurz angerissen wurden.

Um näher zu verstehen, wie die Definition eines Profils innerhalb der UML vor sich geht, wollen wir uns das folgende Beispiel aus der EJB-Welt anschauen. (Abb. 3.17) und an ihm im Folgenden die wichtigsten Konzepte und Mechanismen verdeutlichen.

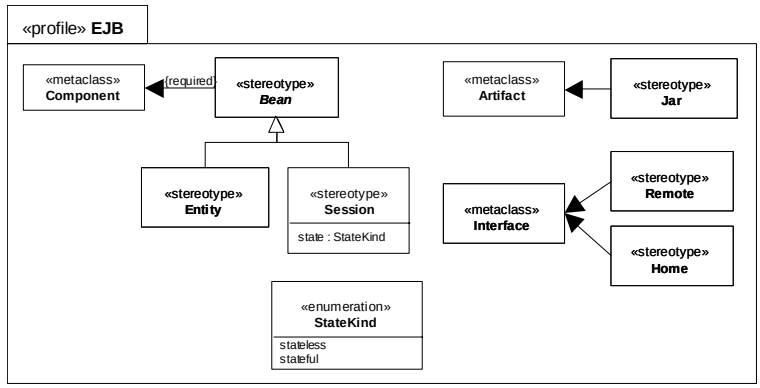


Abb. 3.17 UML-Profil für Enterprise Java Beans (EJB)

Stereotypen

Vorherrschendes Konstrukt zur Erstellung von Profilen ist das Konzept der Stereotypen. Die UML-Spezifikation beschreibt diese als *spezielle Klassen innerhalb des Metamodells, die andere Klassen näher beschreiben und ihnen neue Details hinzufügen.*

Dargestellt werden Stereotypen durch die Einbettung des Stereotypnamens in französische Anführungszeichen (*Guillemets*: „«“, „»“), der dann über dem eigentlichen Klassennamen platziert wird. Ein Element des Typs `Jar` kann so durch die Bezeichnung `«Jar»` kenntlich gemacht, und in Modellen, die das obige Profil importieren, verwendet werden.

Abb. 3.18 Darstellungsalternativen von Stereotypen

Sollen einer Klasse mehrere Stereotypen zugeordnet werden (die Anzahl ist seit UML 2 nicht mehr auf einen Stereotypen beschränkt), so geschieht dies wahlweise durch die Angabe aller Stereotypennamen durch Kommata getrennt, durch das Untereinanderschreiben der einzelnen Stereotypen oder durch die Zuweisung eines graphischen Symbols, das genau diese Zusammenstellung von Stereotypen symbolisiert (Abb. 3.18).

Darstellung von Stereotypen als Klasse

Stereotype sind selbst Klassen und werden somit unter Verwendung der UML-Regeln für Strukturdiagramme definiert und dargestellt. Will man ein Stereotyp Element selbst beispielsweise als Klasse graphisch darstellen, so geschieht dies durch die Kennzeichnung der Stereotypklasse mit dem Stereotyp `«stereotype»` bzw. durch die für alle Stereotypen geltenden Darstellungsalternativen, die im vorigen Abschnitt besprochen wurden. Beispiel für eine solche Darstellung ist der Stereotyp `«Jar»` in der rechten oberen Ecke in Abb. 3.17.

Extensions zur Definition neuer Stereotype

Die grafische Darstellung der Definition eines neuen Stereotyps und die damit verbundene Erzeugung eines neuen Metaelementes geschieht durch das UML-Konstrukt *Extension (dt. etwa: Erweiterung)*. Hierbei handelt es sich um eine spezielle Art der Assoziation in Form eines Pfeils mit ausgefüllter schwarzer Spitze, die vom erweiternden (neuen) Element zum erweiterten (bereits vorhandenen) Element weist. Das erweiterte Element erhält den Namen *Metaklasse* des Stereotyps und kann optional mit dem entsprechenden Stereotyp (`«metaclass»`) versehen werden.

Eine Anwendung dieses Mechanismus ist in Abb. 3.17 in der rechten oberen Ecke mit der Erweiterung der Klasse `Artifact` dargestellt. `Jar` erweitert `Artifact` um zusätzliche Bedingungen, wie zum Beispiel der Notwendigkeit des Vorhandenseins einer

syntaktisch korrekten Manifest-Datei, „erbt“ jedoch auch alle Eigenschaften der zugehörigen Metaklasse *Artefakt*, wie sie in der UML-Spezifikation festgehalten sind. Zu den geerbten Eigenschaften gehören beispielsweise die graphische Darstellung, Regeln zur Verwendung (Artefakte sind immer konkrete Ausprägungen von Komponenten und können nicht selbst instanziiert werden) sowie weitere Besonderheiten, auf die hier nicht näher eingegangen werden soll.

Constraints

Constraints stellen Bedingungen an Modellelemente und werden auch bei der Definition von Profilen rege genutzt, um erweiterte Bedingungen an die Benutzung von Elementen des Metamodells zu stellen.

Wird beispielsweise der Extensionpfeil einer Profildefinition mit dem Constraint `required` versehen, bedeutet dies, dass die Benutzung des Stereotyps – innerhalb eines Paketes auf dem das entsprechende Profil angewendet wird (siehe unten) – obligatorisch ist und die erweiterte Metaklasse nicht direkt benutzt werden darf. Beispiel in Abb. 3.17 ist die Erweiterung von `Component` durch den Stereotyp `«Bean»`. `Component` beschreibt alle technologieunabhängigen Konzepte der komponentenbasierten Entwicklung, `Bean` die Festlegung auf die Technologie *Enterprise Java Beans*. `Bean` selbst ist als abstraktes Element gekennzeichnet und kann daher auch nicht direkt instanziiert werden. Direkt instanziierbar sind `Entity` und `Session`, was Sinn macht, da eine `Bean` in der gewählten Technologie lediglich eine der beiden Ausprägungen besitzen kann – und innerhalb des Beispielprofils auch muss.

Darstellung und Anwendung von Profilen

Die Darstellung eines Profils geschieht durch die Kennzeichnung des UML-Elementes `Package` durch den Stereotyp `«profile»`. Die Metaklasse `Profile` spezialisiert `Package` und erhält dadurch alle Verwaltungsfunktionalitäten des selbigen wie *Merge*, *Import* usw.

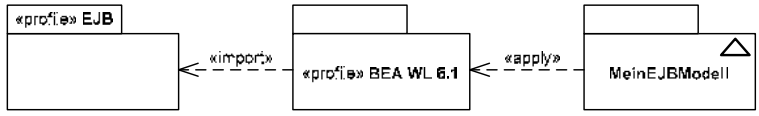


Abb. 3.19 Profilanwendung und –import

Gleichzeitig wird die Funktionalität des Pakets zusätzlich um eine Beziehung namens `ProfileApplication` erweitert, die graphisch mittels einer, mit `«apply»` stereotypisierten, Abhängig-

keitsbeziehung dargestellt wird und anzeigt, dass ein Profil auf ein Paket angewendet wird. Damit darf es alle in dem Profil definierten Konstrukte (Stereotype usw.) verwenden. Die Anwendung eines Profils auf ein Modell mittels dieses Mechanismus zeigt Abb. 3.19. Anmerkung: Ein Modell ist eine Spezialisierung von Paket und wird zur Unterscheidung von diesem mit einem Dreieckssymbol versehen.

Das Beispiel zeigt ein Profil der Enterprise Java Bean Welt mit Namen `BEA WL 6.1`, das speziell zur Verwendung des *BEA Application Servers* erstellt wurde und auf die Elemente des gerade besprochenen EJB Profils aufbaut, indem es diese auf Packageebene importiert. `MeinEJBModell` wendet das so entstandene Profil an und macht die in den Profilen definierten Elemente für sich instanzierbar.

3.4 UML-Repository

Ob als Erweiterung eines bereits bestehenden Metamodells oder völlig auf der grünen Wiese entstanden – wie wir gesehen haben sind alle Modelle der UML letztendlich (direkt oder indirekt) Instanzen von Klassen eines MOF-Metamodells. Es gibt jedoch einen klaren Unterschied zwischen diesen Instanzen und den Diagrammen, mit denen sie visuell dargestellt werden (beispielsweise einem Klassen- oder Aktivitätsdiagramm), der in den Betrachtungen oft unterschlagen wird. Dieser Unterschied ist aber für das Verständnis der UML und unserer weiteren Betrachtungen äußerst wichtig und soll nun näher betrachtet werden.

Eine Bedeutung viele Darstellungsmöglichkeiten

Der UML-Stack stellt, basierend auf der MOF, ein Metamodell zur Spezifikation von Systemen der realen Welt mit den Elementen der UML zur Verfügung. Dazu werden Elemente definiert, die instanziiert und zur Modellierung von System nach genau definierten Regeln kombiniert werden können. Weniger bekannt ist, dass die UML zwar die Instanziierungs- und Kombinationsregeln vorgibt, die tatsächliche visuelle Darstellung in Diagrammform jedoch lediglich eine Empfehlung darstellt. Wird ein Modellelement gemäß der Vorgaben seines Metamodells instanziiert, landet es im UML-Repository, wo es dem schon bestehenden Gesamtmodell hinzugefügt wird und für zukünftige Operationen zur Verfügung steht.

Diagramme sind Sichten auf das UML-Repository

Wie passen nun die UML-Diagramme in dieses Bild? Diagramme sind *Sichten* auf das UML-Repository. Die Visualisierung dieser Sichten geschieht im Falle der UML-Diagramme mittels der durch die OMG definierten graphischen Konstrukte. Der Ansatz kann aber weiter generalisiert werden, indem man anstatt von Diagrammen alle Notationen als Sichten zulässt. Diese Notationen (oder auch Domänen-Spezifischen Sprachen) stellen jeweils besondere Aspekte des im UML-Repository modellierten Systems in ihrer eigenen Weise mit ihrer eigenen Syntax dar. Dieses notationsspezifische Mapping zwischen darstellenden Notationselementen und Bestandteilen des UML-Repositorys muss dabei keine 1-1 Beziehung sein, sondern geschieht kontextspezifisch. Sind Elemente des Repositorys für die betrachtete Domäne gänzlich uninteressant kann es sogar sein, dass sie überhaupt keine Berücksichtigung in der Darstellung finden. Diese Eigenschaft war unter dem Aspekt der geeigneten Abstraktion sogar ausdrücklich erwünscht. Abb. 3.20 verdeutlicht den eben vorgestellten Gedanken noch einmal am Beispiel.

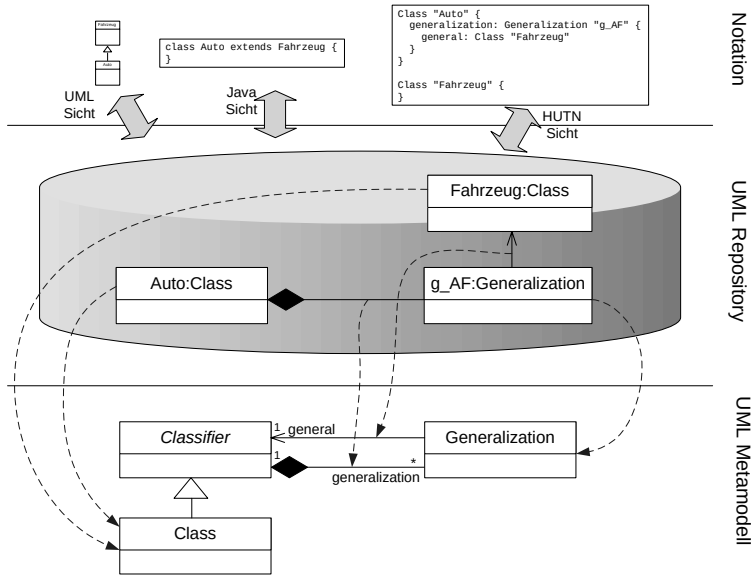


Abb. 3.20 UML-Repository – Ein Beispiel

Die Abbildung zeigt unten einen Ausschnitt aus dem UML-Metamodell (UML-Superstructure) zur Modellierung von *Vererbungsbeziehungen (engl. generalization)*; oben drei verschiedene Notationen des Sachverhaltes in Form eines Klassendiagrammes, einer Java Klasse und als Notation in der Human-Usable Textual Notation [HUTN], einem weiteren Standard der OMG, der eine menschen-lesbarere Alternative zum XMI Format darstellt. Greifen

wir die Notation als Klassendiagramm heraus, ist folgender Sachverhalt dargestellt: `Fahrzeug` ist Superklasse der Klasse `Auto`. In der Mitte der Abbildung ist die notationsfreie „Essenz“ des Modells zu finden: als Instanziierung des UML-Metamodells in Form des Modells im Repository.

Beziehungen zwischen Diagrammen?

Das Wissen über die Existenz des UML-Repositorys- und seine Rolle in der Verbindung zwischen den Sichten der UML in Form der verwendeten Notation, dem Modell des Systems und dem unterliegenden Metamodell, birgt auch die Antwort auf eine oft gestellte Frage nach einer Möglichkeit, verschiedene Modelle und deren Elemente miteinander zu verbinden.

Nehmen wir als Beispiel die UML-Diagramme: Alle Diagramme, die *ein gemeinsames System* beschreiben, besitzen *ein gemeinsames Repository*, in denen die unterliegenden Modellelemente (Instanzen des zugehörigen Metamodells) vorgehalten werden. Ist vom Metamodell eine Beziehung zwischen diesen Elementen erlaubt, kann sie auch modelliert und so die gewünschte Verbindung hergestellt werden. Die Verknüpfung von Modellen verschiedener Diagramme geschieht also, indem die zugehörigen Modellelemente in ein gemeinsames Repository gestellt und dort die notwendigen Verknüpfungen vorgenommen werden. Die Verbindung ist also bei korrekter Benennung und Verbindung der Elemente der einzelnen Diagramme von vorneherein gegeben. Wird ein Verhalten beispielsweise in einem Aktivitätsdiagramm modelliert und soll damit das Verhalten einer Operation innerhalb eines Klassendiagrammes dargestellt werden, ist die Zuordnung eindeutig möglich. Aktivitäten können laut UML-Superstructure Instanzen von `Classifier` zugeordnet werden, was im Falle der Operation (Subklasse von `Classifier`) die Verbindung modellierbar macht, falls sich die Elemente der beiden Diagramme im selben Repository befinden.

Bedeutung und Einsatz

Neben seiner konzeptuellen Bedeutung innerhalb des Mechanismus der UML zur Darstellung von Metamodellen durch die MOF, den wir in den vorigen Abschnitten schon ansatzweise beleuchtet haben, hat das UML-Repository jedoch auch in der praktischen Arbeit mit der UML seinen Platz. Dies gilt vor allem da, wo die UML nicht nur zur Dokumentation verwendet, sondern vollständig und toolgestützt in den Entwicklungsprozess integriert wird. Immer mehr Werkzeuge wie Modellierungstools und Transformatoren gebrauchen eine Implementation des UML-Repository als Speicherort für die mit ihnen verwendeten Modelle. Zwar ist sowieso eine irgendwie geartete Zwischenrepräsentation der Daten notwendig, neu ist jedoch, dass

sich auch hier 1-1 Umsetzungen des UML-Repository Standards durchsetzen und auf proprietäre Modelle weitestgehend verzichtet wird. Dies hat vor allem den Vorteil, dass durch den unterliegenden MOF-Ansatz schnell Änderungen am Metamodell vorgenommen werden können, die Werkzeuge also nur lose an den jeweiligen Standard gekoppelt sind, mit denen sie arbeiten. Dieses Umschwenken ist vor allem auf die massiven Probleme der einzelnen Hersteller zurückzuführen, die mit der Umstellung der UML von Version 1.5 auf 2.0 verbunden waren.

Im Java Umfeld hat sich hier vor allem eine Open-Source Lösung durchgesetzt, die eine direkte Umsetzung des UML 2 Metamodells darstellt, und die wir im praktischen Teil des Buches auch für unsere Zwecke verwenden wollen: das UML2 [EUML2a] Framework des Eclipse Tools Projektes. Einzelheiten dieses Frameworks werden an späterer Stelle erläutert (vgl. Kapitel 8 – Projektplanung), weshalb wir an dieser Stelle nicht näher auf Details eingehen wollen.

3.5 UML-Action-Semantics

Nachdem wir schon in den vorigen Abschnitten immer wieder auf Details aufmerksam gemacht haben, die eine Schlüsselrolle im Gesamtgebäude UML einnehmen, jedoch sehr häufig unerwähnt bleiben, wollen wir auch an dieser Stelle auf ein solches Konstrukt aufmerksam machen: den UML *Action Semantics*. Anstatt jedoch direkt zu beschreiben, was es mit dieser Bezeichnung auf sich hat, wollen wir die Unzulänglichkeiten der Zeit vor ihrer Aufnahme (vor UML 1.5) in den UML-Stack beschreiben und dann zeigen, wie sie aus der Notwendigkeit diese Probleme in den Griff zu bekommen hervorgegangen sind.

Vor UML 1.5 …

Fehlende Elemente zur Beschreibung von Verhalten

Während die Diagramme der UML ausgereifte und zahlreiche Mechanismen bieten, strukturelle Aspekte von Systemen vollständig zu beschreiben, beschränkte sich die Modellierung der dynamischen Aspekte (des Verhaltens zur Laufzeit) im Wesentlichen auf die Darstellung des Aufrufes von Verhalten bzw. der Angabe der Ereignisse, die zu Zustandswechseln oder Verhaltensaufrufen führen. Die Form der Beschreibung dessen, was dann wirklich konkret geschieht (beispielsweise innerhalb einer Operation *nach* ihrem Aufruf – eine White-Box Sicht der Operationen) – wurde nicht vorgegeben. Zwar boten die Diagrammarten zur Modellierung dynamischer Aspekte (bspw. Zustands- oder Aktivitätsdiagramm) Elemente zur Modellierung solcher Sachverhalte an, waren aber lange nicht präzise und vollständig genug, um von wirklicher White-Box-Modellierung

sprechen zu können. Das Füllen dieser Lücke lag somit in der Phantasie des Modellierenden bzw. bei Herstellern der Tools mit denen diese arbeiteten.

In der Praxis wurden und werden zum Schließen der Präzisionslücke zwischen Modell und schließlich resultierendem Quellcode Fragmente der Zielsprache verwendet, die an die entsprechenden Modellelemente (z. B. Methoden) angehängt und bei der Generierung der Anwendung unverändert in den generierten Code eingewoben werden. Von einer plattform-unabhängigen Modellierung ist keine Rede.

Einige Modellierungstools verwenden auch eigene, von konkreten Programmiersprachen abstrahierende Sprachen, die sie dann in jeweilige Zielsprache transformieren. Dies kommt dem eigentlichen Ziel zwar schon näher, stellt jedoch aufgrund der fehlenden Standardisierung nicht nur eine Zwangsbindung an das einmal gewählte Tool dar, sondern bildet aufgrund oft subtiler Unterschiede auch bei der Verwendung ein und derselben Sprache in zwei unterschiedlichen Tools oft eine schwer zu lokalisierende Fehlerquelle.

Dieses Manko der UML war umso schmerzlicher, als gerade die Möglichkeit der präzisen Definition dynamischen Verhaltens –wie zum Beispiel der Änderung von Variablenwerten, der Erzeugung und Zerstörung von Objekten usw. – eine der Grundvoraussetzungen zur effektiven Durchführung eines effizienten modellgetriebenen Entwicklungsansatzes darstellt. Neben der schon erwähnten Probleme bei der Generierung von Quellcode, wird auch die Simulation des Systems am Modell bereits in frühen Stadien der Entwicklung (Stichwort *Executable UML* [MB02]) zur Vermeidung falsch verstandener Anforderungen bzw. dem Ausprobieren alternativer Lösungswege erst durch eine solche Präzisionsstufe der verwendeten Modellierungssprache möglich.

Die Lösung in Form einer Semantikbeschreibung

Die Lösung des beschriebenen Problems lag im Falle der UML darin den geforderten Funktionsumfang einer Sprache zur präzisen Modellierung von Verhalten zu ermitteln und zu beschreiben. Also die Semantik vorzugeben, die konkrete Notation allerdings nicht vorwegzunehmen, sondern die Schaffung der konkreten Syntax den Herstellern der Tools zu überlassen. Auch hier wurde also im besten Sinne der repositoryzentrierten Architektur gehandelt (vergl. Abschnitt 3.4).

In der Notation einer Sprache, die die in der Spezifikation definierten Anforderungen erfüllt (sei sie textuell, graphisch oder sonst wie), ist es dann möglich, zusammen mit den schon vorhandenen Konstrukten der UML-Diagramme ein System so vollständig zu

beschreiben, dass es auf jeder weiteren Plattform, die in ihrer Ausdruckskraft mindestens der UML gleich mächtig ist, realisiert und ausgeführt werden kann.

Seit UML 1.5 …

Die OMG stellt seit Version 1.5 der UML als Teil der Superstructure mit den *ActionSemantics* ein solches Metamodell samt Semantikbeschreibung zur Verfügung und legt damit ein weiteres Stück des Weges zur Realisierung des MDA Gedankens zurück.

Warum nicht einfach Programmiersprachen?

Bevor wir uns jedoch näher mit dem Aussehen und den Mechanismen dieser Sprache beschäftigen, wollen wir noch die Frage beantworten, warum es nicht ausreicht, einfach eine schon bestehende Programmiersprache zu nehmen anstatt etwas völlig Neues zu definieren. Die Antworten decken sich zum größten Teil mit der Argumentation, die bereits in unserer Diskussion der Domain-Specific Languages aufgetaucht ist.

Da der Einwand, den ganzen Aufwand doch sein zu lassen und einfach wie bisher zu programmieren, jedoch immer wieder auftaucht, kann man in den oft heftigen Diskussionen gar nicht zu viele Gegenargumente besitzen, weshalb wir sie an dieser Stelle auch liefern wollen.

Das Verwenden von Programmiersprachen zum Zwecke der plattformunabhängigen Modellierung von Systemverhalten ist *nicht* der goldene Weg, weil

- sie oft mehr Funktionalität als benötigt besitzen, was sie komplexer und damit die resultierenden Modelle für Nicht-Programmierer zu umständlich lesbar macht. *Keep it simple stupid.*
- Die Verwendung einer konkreten Zielsprache oft schon den Gebrauch dort oft gebrauchter Konstrukte und Lösungsansätze (Idiome) bei der eigentlich „untechnischen“ Modellierung des erwünschten Verhaltens suggeriert und so eventuell effizientere Lösungen als die Altgewohnten „übersehen“ werden.
- Direkt ans Modell angetragene Code-Fragmente eine Erosion des *Separation of Concern* Gedankens darstellen. Tatsächliche fachliche Anforderungen an das System können nur noch schlecht von Elementen unterschieden werden, die lediglich zur Unterstützung einer konkreten Plattform benötigt werden, mit den altbekannten aus einem solchen Umstand resultierenden Problemen in der Kommunikation zwischen Fachlichkeit und Technik.

Nach dieser Motivation der Einführung der ActionSemantics (AS) in den UML-Stack wird es nun Zeit, uns am konkreten Beispiel einmal anzusehen, wie die präzise Modellierung dynamischen Verhaltens mittels Elementen der AS denn nun wirklich aussieht. Wir wollen dazu ein Beispiel aus einer Artikelserie von Conrad Bock heranziehen [Boc03b], die einen hervorragenden Überblick über Aktivitätsdiagramme im Allgemeinen bietet und dabei auch die AS streift.

Modelliertes Verhalten am Beispiel

Im Folgenden werden die Kenntnis der Notation und Semantik von Aktivitätsmodellen zum größten Teil vorausgesetzt und an dieser Stelle nicht näher besprochen. Wer sein Wissen zuvor kurz auffrischen will, der wird in Anhang A fündig, wo alle benutzten Konstrukte aufgeführt und kurz erläutert werden.

Das Beispiel modelliert ein Verhalten, in dem ein Haus gestrichen werden soll anhand der Aktion `Streiche Haus`, die als Input die `Neue Farbe` und das `Haus`, das die Farbe wechseln soll, bekommt und dann präzise den darzustellenden Vorgang des Farbwechsels – hier ein Inhaltswechsel einer Variable – darstellt.

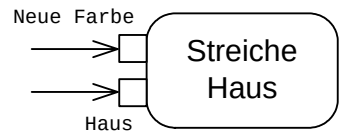


Abb. 3.21 Das Beispiel in der Notation als Aktion (nach [Boc03b])

Abb. 3.21 zeigt das Beispiel notiert als Aktion `Streiche Haus`. Zugegebenermaßen ist das Modell auf den ersten Blick wenig eindrucksvoll: bis auf den Namen der Aktion und den beiden Eingangsparametern bietet es anscheinend keine weiteren Hinweise auf das, was wirklich passieren soll. Wo ist die vorher so angepriesene Präzision also geblieben?

Auch hier steckt die Antwort in einer tieferen Schicht als sie die Diagramme bilden: dem bereits in 3.4 vorgestellten UML-Repository. Abb. 3.22 zeigt den relevanten Ausschnitt des unterliegenden UML-Repositorys: In der rechten oberen Ecke die Aktion `Streiche Haus` als Instanz der Klasse `AddStructuralFeatureValueAction`, die wie alle vordefinierten Aktionen der UML von der Klasse `Action` erbt und in der Superstructure definiert ist.

Mit den Eigenschaften ihrer Basisklasse erbt `Streiche Haus` auch die Definition der dort hinterlegten Semantik: `AddStructuralFeatureValueAction` fügt dem durch den `InputPin object` spezifizierten Element das Attribut `value` hinzu bzw. ändert den dort gespeicherten Wert, falls das Attribut bereits besteht und `replaceAll` auf `true` gesetzt ist.

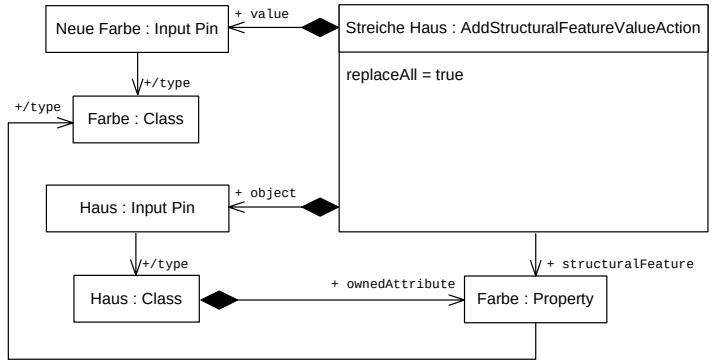


Abb. 3.22 Die Aktion „Streiche Haus“ im Repository (nach [Boc03b])

Mit dieser Beschreibung haben wir jedoch – zugegebenermaßen auf sehr abstrakter Ebene – genau den Sachverhalt ausgedrückt, den wir ursprünglich modellieren wollten: die Änderung des Attributwertes eines Objektes zur Laufzeit.

Aktionen als Basis allen Verhaltens

In der UML ist es nur Aktionen (also Subklassen der Klasse `Action`) erlaubt, zur Laufzeit Änderungen an Modellen vorzunehmen. Jedes Verhalten, das tatsächlich „etwas bewirkt“ lässt sich also letzten Endes auf eine Aktion zurückführen. Aus diesem Grund werden sie manchmal auch die *Primitive der Verhaltensmodellierung* genannt.

Alle zur Verhaltensmodellierung benötigten Aktionen stellt die UML innerhalb der UML-Superstructure zur Verfügung. Der Anwender der UML kann sie nach Belieben zur Darstellung von Verhalten (*engl. Behavior*) kombinieren, um so das betrachtete System abzubilden.

Wie beschrieben, ist die konkrete Notation von Aktionen nicht vorgegeben und kann so durch die verwendeten Tools bestimmt werden. Direkt zum Einsatz kommen Aktionen beispielsweise im Aktivitätsdiagramm, wo sie die Grundbausteine von Aktivitäten bilden.

3.6 OCL – Object Constraint Language

Wie wir im vorigen Abschnitt gesehen haben, bietet die UML Action Semantics die Möglichkeit, Modelle durch die Verfeinerung der Beschreibung ihrer dynamischen Bestandteile zu präzisieren, indem sie modellierbar macht, *wie* etwas passieren soll, also die White-Box Sicht auf Verhalten ermöglicht. Eine Möglichkeit Modelle derart zu präzisieren, dass zwar klar ist, *was* gemacht werden soll, das *Wie* aber explizit ausgeklammert wird, werden wir in diesem Teil des Kapitels kennen lernen. Dazu wollen wir auch an dieser Stelle zuerst motivieren, wofür der Standard gebraucht wird und dann eine kurze Einführung in seinen Gebrauch bieten.

Warum noch ein Standard?

Obwohl die Diagramme der UML bereits eine Menge der notwendigen Mittel zur Verfügung stellen, die man für eine ernsthafte Anwendung der MDA zur Modellierung, bieten sie allein dennoch nicht die notwendige Präzision, die der Ansatz von einer Modellierungssprache fordert. Formale Sprachen zur Systemmodellierung (z. B. Object-Z [Gra00]) wiederum sind zwar höchstpräzise in ihrer Darstellung, haben sich als alleiniges Mittel zur Modellierung bis heute allerdings nicht durchsetzen können. Dies liegt vor allem daran, dass ihnen die visuelle Komponente fehlt, die eine diagrammatische Modellierung bietet, und deshalb die Übersichtlichkeit und Wartbarkeit der Modelle leidet. Zusätzlich bieten sie oft nicht die Mittel zur ausreichenden Abstraktion der betrachteten Systeme, wie wir gesehen haben, ist dies auch eine der Hauptanforderungen an die verwendete Sprache.

Als Kompromiss bietet sich ein hybrider Ansatz aus textueller und visueller Modellierung an. Die *Object Constraint Language (OCL)* [OCL2], ein Bestandteil der OMG-Standards, bietet als Ergänzung zum diagrammatischen Teil der UML 2 die Möglichkeit, *Ausdrücke (engl. expressions)* an die Modellelemente anzutragen, die die noch fehlenden Details beschreiben und so die Präzisionslücke zwischen (Software-)System und Modell weitestgehend zu schließen.

3.6.1 Grundlagen – Was ist OCL?

OCL ist eine streng typisierte Sprache. Diese Typisierung erlaubt die Auswertung der OCL-Syntax bereits zur Modellierungszeit, welche ansonsten erst mit Vorliegen einer (lauffähigen) Modellinstanz geschehen könnte und die Anwendung unnötig verkomplizieren würde.

Sie basiert auf den mathematischen Konzepten der Mengenlehre und Prädikatenlogik und bildet eine Untermenge der aus der mathematischen Symbolschrift bekannten Funktionalität. Auf die Übernahme der mathematischen Symbole selbst wurde allerdings verzichtet und diese durch eine eigene formal textuelle ASCII-Notation ersetzt. So wird versucht, eine Balance zwischen bewährter aber für den Gebrauch außerhalb der Mathematik zu komplexer formal mathematischen Beschreibung und der Verwendung zwar eingängiger, jedoch für die Detaillierung von Systemen zur Verwendung in MDA Ansätzen zu unpräziser natürlichsprachlicher Prosaspezifikationen zu finden.

Die Detaillierung der UML-Modelle mittels OCL geschieht, indem *Bedingungen (engl. constraints)* an die betroffenen Elemente angetragen werden. Dieses können vollwertige *Ausdrücke (engl. expressions)* sein, die beispielsweise den gültigen Wertebereich von Variablen und Sachverhalten eines (Teil-)modells einschränken (`x<5`, `x=5`, `WENN y=3 dann x UNGLEICH 4`), aber auch komplexere Konstrukte wie die Angabe initialer Modellelementzustände (Variablenwerte, Überprüfung des Vorhandenseins von Objekten und Assoziationen), die Beschreibung von Abfragen, die als Ergebnis Teilmengen des unterliegenden Modells zurückliefern, usw.

… und was nicht.

OCL ist rein deklarativ

Obwohl sie auf den ersten Blick aufgrund ihrer textuellen Notation vielleicht den Eindruck erweckt, ist die OCL kein Ersatz für eine imperative Programmiersprache, sondern besitzt einen rein deklarativen Charakter.

Deklarativ heißt, sie beschreibt lediglich die Bedingungen, die die Instanz eines Modells erfüllen muss, um als gültig zu gelten (das Ergebnis einer Operation, das Vorhandensein bestimmter Modellelemente, ein dezidierter Variablenwert, …). Sie gibt nicht vor, wie das Modell in diesen gültigen Zustand gelangt bzw. was passiert, wenn eine Bedingung verletzt wird. Solche Beschreibungen fallen unter die Kategorie imperativer Konstrukte, die beispielsweise von der UML ActionSemantics abgedeckt wird (siehe Abschnitt 3.5) und hier nicht erneut aufgeführt wird.

Durch diese Vorgehensweise wird die größtmögliche Präzision der Systembeschreibung unter gleichzeitiger Wahrung eines hohen Abstraktionsniveaus garantiert, das eine Vorgabe von Implementierungsdetails bereits in der Analyse verhindert. Eine solche Vorgabe würde die tatsächliche Realisierungsphase des Systems unnötig vorwegnehmen und eine effiziente Durchführung des Gesamtprozesses be- wenn nicht sogar verhindern.

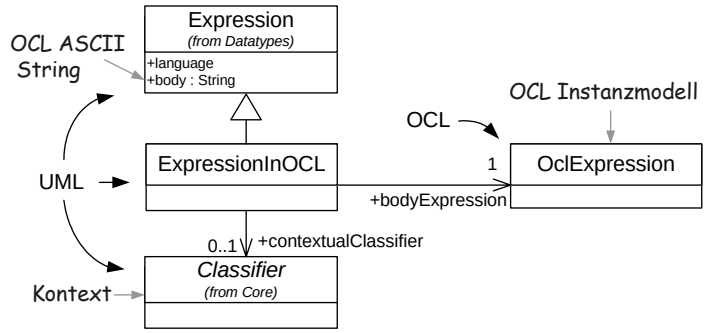
3.6.2 Zuordnung von OCL-Ausdrücken zu Modellelementen

Grundsätzlich existieren drei Möglichkeiten OCL-Ausdrücke mit UML-Elementen zu verknüpfen. Wir wollen mit der abstraktesten der Drei anfangen und uns dann zu den anwenderfreundlicheren vorarbeiten, die auch die sind, mit denen man als Anwender am häufigsten in Kontakt kommt.

Zuordnung im Metamodell/UML-Repository

OCL-Ausdrücke sind immer eindeutig einem dezidierten Modellelement zugeordnet, das der *Kontext (engl. context)* des Ausdruckes genannt wird. Laut Spezifikation [OCL2] ist hier jedes Element vom Typ `Classifier` möglich und kann somit eine Klasse, eine Komponente, eine Operation, eine Aktivität o. Ä. sein. Abb. 3.23 zeigt diesen Sachverhalt anhand eines Ausschnittes aus der OCL-Spezifikation.

Abb. 3.23 Ausschnitt aus der OCL-Spezifikation



`ExpressionInOCL` repräsentiert den OCL-Ausdruck und erbt von `Expression` aus der UML-Kernspezifikation. In `language` wird der Typ `'OCL'` vermerkt, in `body` steht die textuelle Notation des Ausdrucks, für die wir im Folgenden noch einige Beispiele sehen werden. Die Repräsentation des Ausdrucks als Instanz der abstrakten Syntax des OCL Metamodells, das mittels MOF spezifiziert wurde (vgl. hierzu Abschnitt 3.4), steht als Assoziation unter `bodyExpression` zur Verfügung.

Das Modellelement selbst, für das der Ausdruck gilt, ist unter `contextualClassifier` zu finden. Die Konstrukte `Expression`, `ExpressionInOCL` und `Classifier` stammen dabei aus der UML-Kernspezifikation. `OCLExpression` ist Teil der OCL-Spezifikation.

Nun wissen wir also, wie Instanzen von OCL-Ausdrücken im Metamodell – und damit zur Laufzeit auch im Repository – UML Elementen zugeordnet werden. Dieses Wissen ist vor allem dann interessant, wenn man direkt auf dem Repository arbeiten muss, z. B. bei der Erstellung entsprechender Tools innerhalb des eigenen MDA Frameworks. Noch ist offen, wie sich eine solche Zuordnung in den entsprechenden Diagrammen niederschlägt, um OCL auch in der Modellierung selbst benutzen zu können.

OCL in Diagrammen

In Diagrammen wird eine hybride Form aus UML-Elementen und textueller Repräsentation benutzt, die so aussieht, dass textuelle OCL-Ausdrücke mittels Kommentar an das betroffene Element (den Kontext) angehangen werden – Abb. 3.24 zeigt dies am Beispiel.

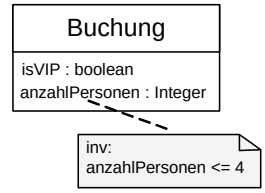


Abb. 3.24 OCL in Diagrammen

Modelliert ist die Klasse `Buchung`, die wir im weiteren Verlauf noch näher kennen lernen werden, an dieser Stelle aber nicht näher erläutert werden soll. Wichtig an dieser Stelle ist lediglich, dass mittels OCL spezifiziert wurde, wie viele Personen pro Buchung erfasst werden dürfen.

Dazu wurde an das betroffene Modellelement (hier das Attribut `anzahlPersonen`) eine Invariante (`inv`) angehangen, die zur Laufzeit des Modells zu keinem Zeitpunkt verletzt werden darf und so den Wert des Attributs auf höchstens vier beschränkt.

OCL „pur“

Neben der Repräsentation im Repository, die natürlich immer vorhanden ist, ist die Angabe von OCL-Ausdrücken noch in einer weiteren Form möglich, ohne das betroffene Element gleichzeitig in einer Diagrammform angeben zu müssen.

```
context Buchung
inv: anzahlPersonen <= 4
```

Abb. 3.25 OCL – textuelle Repräsentation

Ist dies gewünscht, findet die rein textuelle Repräsentation Verwendung, in der dann allerdings explizit und eindeutig der Kontext des

Ausdruckes angegeben werden muss. Das vorige Beispiel in dieser Variante sähe dann wie in Abb. 3.25 dargestellt aus.

Einzige Bedingungen an den Kontext ist auch hier, dass es sich um eine Spezialisierung (Subklasse) des Typs `Classifier` handeln (hier mit `Class` erfüllt) und sich das Elemente an der spezifizierten Stelle (Paket) im UML-Repository befinden muss.

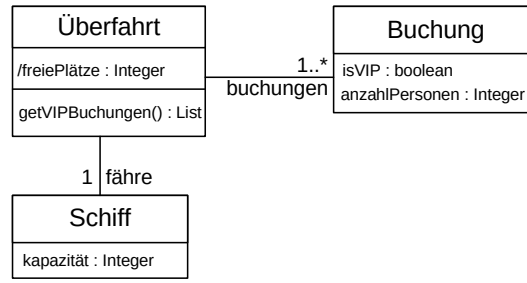
3.6.3 Anwendungsmöglichkeiten von OCL

Abschließend zu diesem Exkurs zur Object Constraint Language schauen wir uns noch kurz an Beispielen an, wofür die Verwendung von OCL grundsätzlich geeignet ist bzw. welche Sprachkonstrukte (eines haben wir mit der Invariante `inv` bereits kennen gelernt) uns die Spezifikation dafür zur Verfügung stellt.

Beschreibung von Geschäftsregeln

Eine weitere Anwendung des Invarianten Konstruktes (`inv`) ist die Definition von Geschäftsregeln. Abb. 3.26 zeigt zur Veranschaulichung ein zugegebenermaßen vereinfachtes Domänenmodell aus dem Kreuzfahrtgeschäft, das uns im weiteren Verlauf des Abschnittes noch öfter begegnen wird und zusätzlich einige weitere Eigenarten der OCL veranschaulichen soll:

Abb. 3.26 Klassendiagramm des Beispiels



Zentrale Klasse ist `Überfahrt`, zu der beliebig viele können, mindestens jedoch eine `Buchung` gehören muss. Buchungen bestehen aus der Angabe der Anzahl Personen, die sie repräsentieren, und halten zudem fest, ob es sich um eine VIP-Buchung handelt. Überfahrten finden traditionell auf genau einem `Schiff` statt. Zu diesem wird festgehalten, welche Kapazität es besitzt (`kapazität`).

Zur besseren Übersichtlichkeit wollen wir die Geschäftsregel selbst wieder in der diagramm-unabhängigen Notation angeben (Abb. 3.27):

```
context Überfahrt
inv: freiePlätze >= - fähre.kapazität / 100
```

Abb. 3.27 Geschäftsregel als Invariante

Kontext ist die Klasse `Überfahrt`, für die wir näher spezifizieren wollen, wie viele Personen insgesamt an ihr teilnehmen können, bevor die Kapazität des zugeteilten Schiffes erschöpft ist. Dazu wird eine Invariante auf dem Attribut `freiePlätze` definiert, die besagt, dass die Überfahrt nur zu einem Prozent der Kapazität des zugeordneten Schiffes überbucht werden darf.

Spezifizierung von Interfaces und Operationen

In der Analyse- und Designphase oft benötigtes Mittel sind Konstrukte zur Beschreibung von Schnittstellen-/ Operationsverhalten. Während mittels der UML mittlerweile Dynamik als White-Box Verhalten beispielsweise mittels Aktivitätsdiagrammen modellierbar ist, bietet OCL durch die Möglichkeit Vor- und Nachbedingungen an Verhaltensaufrufe anzutragen Mittel und Wege das Black-Box Verhalten eines Verhaltensaufrufes, beispielsweise direkt an ein Interface im Diagramm, anzuhängen.

Betrachten wir dazu das Interface in (Abb. 3.28), das eine Operation `div` enthält, die der Division von ganzen Zahlen entsprechen soll. Bis auf den Hinweis im Namen der Operation, geht diese Information zu diesem Zeitpunkt allerdings nicht aus dem Modell hervor.

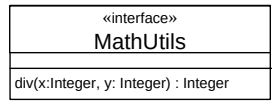


Abb. 3.28 Beispielinterface

OCL bietet nun die Möglichkeit das Verhalten der Operation als Black-Box anzugeben, ohne die Realisierung vorwegzunehmen. Betrachten wir dazu Abb. 3.29:

```
context MathUtils::div(x:Integer, y: Integer): Integer
pre: y <> 0
post: result = x / y
```

Abb. 3.29 OCL-Ausdruck zur Präzisierung

Angegeben ist als Kontext die Operation `div` des Interfaces sowie die Vor- und Nachbedingungen, die gültige Verhaltensaufrufe (und

damit die implementierenden Methoden) zu erfüllen haben. Als Vorbedingung (`pre`) ist gefordert, dass der Divisor `y` der Division nicht null sein darf. Mittels Schlüsselwert `result` wird in der Nachbedingung (`post`) der zurückgelieferte Wert der Operation näher bestimmt und damit die Beschreibung des von außen sichtbaren Verhaltens vervollständigt.

Präzisierung von Attributen

Weitere Möglichkeit des OCL Einsatzes ist die Anreicherung von Diagrammen durch die Angabe von Attributdetails. So können beispielsweise der Initialwert eines Attributes zur Zeit der Instanziierung (Schlüsselwort `init`), oder die genaue Regel angegeben werden, die bestimmt, wie sich ein Wert eines abgeleiteten Attributes ermittelt. So könnte beispielsweise auch angegeben werden, wie sich der Wert des Attributes `freiePlätze` in `Überfahrt` (Abb. 3.26) aus den anderen Attributwerten ermittelt (Abb. 3.30).

Die unterliegende Annahme unterscheidet hierbei zwischen VIPs und Nicht-VIPs. Wie im echten Leben zählen VIPs doppelt, sodass ein VIP zwei Plätze eines Nicht-VIPs belegt. Dass sich die Anzahl der bereits vergebenen Plätze aus Anzahl der Nicht-VIPs addiert zu der zweifach zählenden Anzahl VIPs ergibt. Die noch freien Plätze wiederum sind die Differenz zwischen Kapazität und bereits vergebener Plätze. Dass diese Differenz nicht größer werden darf als ein Prozent der Gesamtkapazität eines Schiffes, bestimmt bereits die Regel aus dem Beispiel zur Beschreibung von Geschäftsregeln (siehe oben).

Abb. 3.30 Beispiel einer Ableitungsregel

```
context: Überfahrt::freiePlätze : Integer
derive: fähre.kapazität
        -
        (buchungen.select(not isVIP)  -- Menge aller nicht VIPs
             collect(anzahlPersonen). -- Menge aller Personenzahlen
             sum()                    -- Summe der Personenzahlen
         +
         buchungen.select(isVIP).     -- Menge aller VIPs
             collect(anzahlPersonen). -- Menge aller Personenzahlen
             sum()                    -- Summe der Personenzahlen
             * 2)                     -- VIP Summe zählt doppelt
```

Die gezeigte Ableitungsregel (Schlüsselwort `derive`) bildet nun den eben beschriebenen Sachverhalt ab (Abb. 3.30). Beginnen wir mit der Beschreibung der Berechnung der bereits vergebenen Plätze im zweiten Teil des Ausdrucks: Betrachtet wird die Menge `buchungen` an `Überfahrt`. Der `select` Operator, der auf alle Mengentypen angewendet werden kann, greift sich alle Elemente der entsprechenden Menge heraus, für die die in ihm angegebene

Bedingung greift. In unserem Falle bestehen die Elemente der Menge aus Buchungen mit Attribut `isVIP`. Es werden somit alle Buchungen herausgegriffen, die den Wert `false` in `isVIP` besitzen, also keine VIPs sind. `collect` greift sich nun alle Werte aus den gefilterten Buchungen heraus, die in der Variable `anzahlPersonen` gespeichert sind; `sum()` schließlich bildet die Summe all dieser Werte, sodass an dieser Stelle des Ausdrucks die Gesamtzahl aller Nicht-VIPs zum Zeitpunkt der Auswertung steht.

Diese Anzahl wird nun zu der Gesamtzahl VIPs addiert, die im zweiten Teil des Ausdrucks ermittelt wird. Die Ermittlung geschieht fast identisch zu der im ersten Teil und unterscheidet sich lediglich dadurch, dass die ermittelte Summe zum Schluss verdoppelt wird.

Diese Gesamtzahl an Personen wird nun von der Kapazität des der Überfahrt zugeordneten Schiffes (`fähre`) abgezogen und so die Anzahl der freien Plätze ermittelt.

Spezifizierung von Abfragen

Eine weitere Eigenschaft der OCL, die sie für die Anwendung von MDA interessant macht, ist die, dass sie dieselbe Mächtigkeit besitzt wie SQL (vgl. [AB01]), also eine vollwertige Abfragesprache darstellt. Diese Eigenschaft kann beispielsweise verwendet werden, um

- den Rückgabewert von Abfrageoperationen wie `findeAuftragByKundennummer(…) : List` mit direktem Bezug zum (fachlichen) Klassenmodell zu spezifizieren,
- Quellbereiche von Transformationen anzugeben (dies geschieht beispielsweise auch im QVT Standard der OMG (vgl. Kapitel 5 Transformation)),
- OCL-Ausdrücke wie Invarianten auf komplette (Teil-)Mengen des Modells auszuweiten,
- usw.

Die Angabe der Operation `getVIPBuchungen():List` aus Abb. 3.26 könnte dann mit den bereits aus den vorigen Beispielen bekannten OCL Sprachelementen wie folgt präzisiert werden (Abb. 3.31):

```
context: Überfahrt::getVIPBuchungen() : List
post: result = buchungen.select(isVIP)
```

Abb. 3.31 Detaillierung einer Abfrageoperation

Weitere Anwendungen und Literaturtipps

Obwohl dieser Abschnitt von vorneherein nur als „Appetitmacher“ konzipiert ist und schon aus Platzmangel nicht ansatzweise alle möglichen Anwendungen der OCL abdecken kann, wollen wir Ihnen einige weitere Anwendungsbeispiele nicht vorenthalten. Die Weiterverfolgung der Ansätze überlassen wir jedoch dem interessierten Leser selbst. Weitere Möglichkeiten sind

- die Beschreibung von Guardbedingungen an Transitionen,
- die Definition von Sprachen (bspw. der abstrakten Syntax von DSLs),
- die Präzisierung von Stereotypdefinitionen durch Invarianten auf die Typen auf die diese angewendet werden dürfen,
- die Angabe gültiger Empfänger für Nachrichten (messages) und Aktionen (actions),
- …

Das Standardwerk zur OCL 2 zur Drucklegung dieser Publikation ist das Buch von Jos Warner und Anneke Kleppe [WK03], das jedem der sich näher mit der Materie auseinander setzen möchte, wärmstens ans Herz gelegt wird. Die Originalspezifikation selbst ist mit vielen Hintergrundinformationen ausgestattet, die nur hier auf dem aktuellsten Stand zu bekommen sind [OCL2]. Weiterhin existieren im Netz aktuell eine Reihe von Projekten, die sich der Realisierung und Einbindung von OCL in Java Programme widmen. Hier ist der Einsatz einer Suchmaschine gefragt, um eine vollständige Liste zu erhalten.

Literatur

[AB01] David H. Akehurst und Behzad Bordbar: *On Querying UML Data Models with OCL*. In: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools. Springer, 2001.

[Amb03a] Scott W. Ambler: *Examining the Model Driven Architecture (MDA)*.
http://www.agilemodeling.com/essays/mda.htm (letzter Abruf Mai 2005)

[BL74] Walter S. Brainerd und Lawlrence H. Landweber: *Theory of Computation*. John Wiley & Sons, 1974.

[Boc03b] Conrad Bock: *UML 2 Activity and Action Models Part 2 – Actions*. In: Journal of Object Technology, Volume 2, Number 5, S. 41-56, ETH Zürich, September/Oktober 2003.
http://www.jot.fm/issues/issue_2003_09/column4/ (letzter Abruf Mai 2005)

[Boo93] Grady Booch: *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 1993.

[CE00] Krysztof Czarnecki und Ulrich W. Eisenecker: *Generative Programming – Methods, Tools and Applications*. Addison-Wesley, 2000.

[CWH00] Mary Campione, Kathy Walrath und Alison Huml: *The Java Tutorial, Third Edition*. Addison-Wesley, 2000.
http://java.sun.com/docs/books/tutorial/ (letzter Abruf Mai 2005)

[DBC] Wikipedia: *Design by Contract*.
http://en.wikipedia.org/wiki/Design_by_contract (letzter Abruf Juli 2005)

[DSL] Wikipedia: *Domain-Specific Language*. http://en.wikipedia.org/wiki/Domain_Specific_Language (letzter Abruf Mai 2005)

[EUML2a] Eclipse.org: *The Eclipse UML2 Project*. http://www.eclipse.org/uml2 (letzter Abruf Mai 2005)

[Fow04a] Martin Fowler: *UmlAsSketch*. http://www.martinfowler.com/bliki/UmlAsSketch.html (letzter Abruf Mai 2005)

[Fow04b] Martin Fowler: *UnwantedModelingLanguage*. http://www.martinfowler.com/bliki/UnwantedModelingLanguage.html (letzter Abruf Mai 2005)

[GDB03] Timothy J. Grose, Gary C. Doney und Steven Brodsky: *Mastering XMI – Java Programming with the XMI, XML, and UML*. John Wiley & Sons, 2003.

[GP] Krysztof Czarnecki und Ulrich W. Eisenecker: *Generative Programming*. http://www.generative-programming.org/ (letzter Abruf Mai 2005)

[Gra00] Graeme Smith: *The Object-Z Specification Language – Advances in Formal Methods*. Kluwer, 2000.

[Har87] David Harel: *Statecharts – A visual Formalism for Complex Systems*. In: Science of Computer Programming, Volume 8, Issue 3, S. 231-274, Elsevier, Juni 1987.

[HUTN] Object Management Group (OMG): *Human-Usable Textual Notation (HUTN) v1.0*. http://www.omg.org/docs/formal/04-08-01.pdf (letzter Abruf Mai 2005)

[JCJO92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson und Gunnar Overgaard: *Object-Oriented Software Engineering – A Use Case Driven Approach*. Addison-Wesley, 1992.

[JMI] Sun Microsystems: *Java Metadata Interface (JMI)*. http://java.sun.com/products/jmi/ (letzter Abruf Mai 2005)

[Lad03] Ramnivas Laddad: *AspectJ in Action – Practical Aspect-Oriented Programming*. Manning, 2003.

[MB02] Stephen J. Mellor und Marc J. Balcer: *Executable UML – A Foundation for Model Driven Architecture*. Addison-Wesley, 2002.

[MOF2] Object Management Group (OMG): *Meta Object Facility (MOF) 2.0 Core Final Adopted Specification*.
http://www.omg.org/docs/ptc/03-10-04.pdf (letzter Abruf Mai 2005)

[MOF2XMI] Object Management Group (OMG): *MOF 2.0/XMI Mapping Specification, v2.1*.
http://www.omg.org/docs/formal/05-09-01.pdf (letzter Abruf September 2005)

[OCL2] Object Management Group (OMG): *UML 2.0 OCL Final Adopted Specification*.
http://www.omg.org/docs/ptc/03-10-14.pdf (letzter Abruf Mai 2005)

[ORe04] Graham O'Regan: *Introduction to Aspect-Oriented Programming*.
http://www.onjava.com/pub/a/onjava/2004/01/14/aop.html (letzter Abruf Mai 2005)

[Par72] David L. Parnas: *On the Criteria to Be Used in Decomposing Systems into Modules*. In: Communications of the ACM, Volume 15, Issue 12, S. 1053-1058, ACM, Dezember 1972.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy und William Lorensen: *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[UML] Object Management Group (OMG): *UML Resource Page*.
http://www.omg.org/uml/ (letzter Abruf Mai 2005)

[WK03] Jos Warmer und Anneke Kleppe: *The Object Constraint Language – Getting Your Models Ready for MDA, Second Edition*. Addison-Wesley, 2003.

[XML] World Wide Web Consortium (W3C): *Extensible Markup Language (XML)*.
http://www.w3.org/XML/ (letzter Abruf Mai 2005)

4 Modelle der MDA

„Diagrams that convey little or no information do more harm than good. It is important not to let important design information drown in a sea of trivialities, especially when large systems are constructed."

Bo Sanden

Im vorigen Kapitel wurden unter anderem die drei Hauptziele der MDA vorgestellt:

- Portabilität,
- Interoperabilität
- und Wiederverwendbarkeit.

Hauptmechanismus der MDA um diese Ziele zu erreichen ist die strikte Verfolgung der Idee die Spezifikation eines Systems von der technischen Realisierung zu trennen. Dazu werden Systeme mittels Modellen zuerst völlig technologieneutral beschrieben und dann nach und nach mit technischen und fachlichen Details angereichert, bis am Ende eine lauffähige Anwendung zur Verfügung steht.

Den Weg, den ein Modell durch die verschiedenen Abstraktionsstufen nimmt – quasi seinen Lebenszyklus – wollen wir im folgenden Kapitel nachzeichnen und die einzelnen Stationen näher betrachten. Zur Veranschaulichung dient ein Modell eines Fallbeispiels, an dem mittels ausgewählter Sichten die theoretischen Überlegungen praktisch erläutert werden.

4.1 Lebenszyklus von MDA-Modellen

Wir wollen nun die einzelnen Stationen näher erkunden, die ein Modell in einem MDA-Entwicklungsprozess zu durchlaufen hat. Abb. 4.1 zeigt eine Übersicht, die uns im Folgenden als Wegweiser dient:

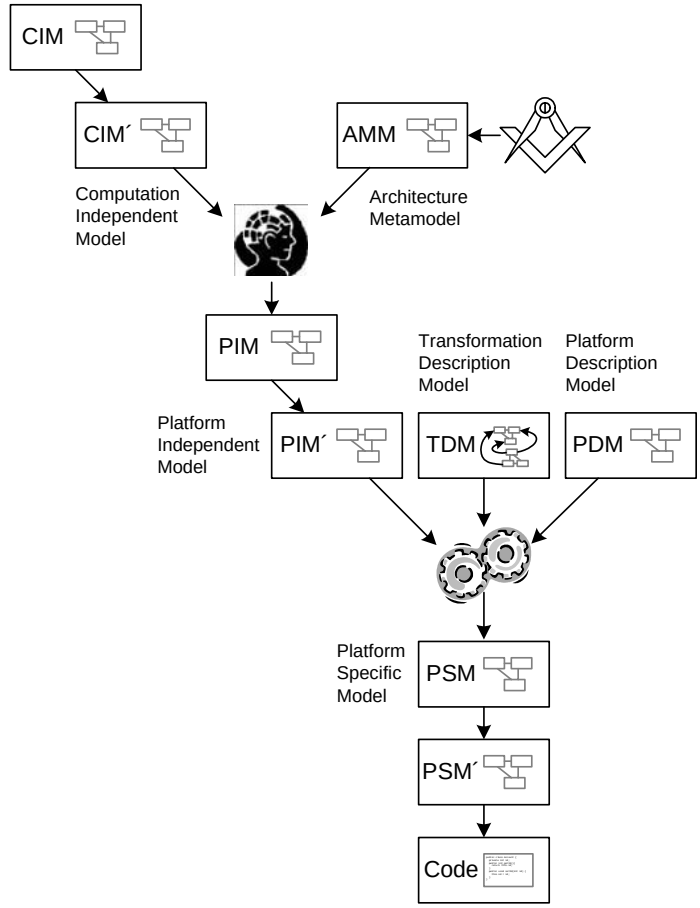


Abb. 4.1 Lebenszyklus von MDA Modellen

Beginnend oben links mit der Klasse der anwendungs-unabhängigen Modelle (CIM), sinkt der Abstraktionsgrad mit jeder dargestellten Stufe, bis das Modell schließlich das Ende des Zyklus erreicht, das durch die fertige Anwendung gebildet wird.

An zwei Stellen wurde die Art des Übergangs von einer Abstraktionsstufe in die Nächste explizit dargestellt. Oben dargestellt die Transformation des CIM in das PIM, die als Eingabe – neben dem CIM selbst – ein so genanntes *Architektur Metamodell (engl. Archi-*

tecture Metamodel, AMM) benötigt, in dem der Zusammenhang der architekturellen Elemente (Konzepte, Muster, Abstraktionen und Schnittstellen) festgehalten ist (→Zirkel-/Winkelsymbol). Es beschreibt damit in formaler Weise so etwas wie einen (eigenen) Architekturstil. Die gemeinte Architektur ist an dieser Stelle jedoch keine technologische, sondern eine domänen-spezifische.

Anmerkung: Der Begriff des Architecture Metamodel taucht in dieser Form nicht in der OMG-Terminologie auf. Unserer Auffassung nach beschreibt er aber den Kern eines solchen Metamodells recht gut. Ein etwas standardkonformerer Begriff ist der des *Platform Independent Metamodel (PIMM)*, der zwar ebenfalls nicht Teil des offiziellen MDA-Jargons ist, er stellt aber auch keinen vollständig neuen Terminus technicus dar. Im Gegensatz zum AMM unterstreicht er die Nähe zum PIM, trifft aber ansonsten keinerlei Aussage zum Inhalt dieses Metamodells. Näheres zu dem Konzept des AMM werden wir in Kürze erfahren.

Der erwähnte Übergang in die plattform-unabhängige Welt des PIM erfolgt manuell und impliziert einen stark kreativen Vorgang (→Kopfsymbol), der das Wissen umfasst wie der erwähnte Architekturstil auf die Problemdomäne anzuwenden ist.

Nach erfolgter Transformation befinden wir uns bereits in der Klasse von Modellen, die anwendungs-spezifische Aspekte in den Vordergrund stellen, jedoch immer noch so weit wie möglich von technologie-spezifischen Details abstrahieren.

Es folgt der zweite explizite Übergang, der diesmal vorzugsweise ganz oder teilweise automatisiert durchgeführt werden sollte, was durch das Zahnradsymbol symbolisiert wird. Hier erfolgt der Übergang von einer technologieneutralen hin zur technologie-spezifischen Modellierung in Bezug auf eine dezidierte Plattform. Synonym wird bei diesem Übergang von Technologie-Projektion bzw. Technologie-Bindung gesprochen.

Technologie-Projektion/ -Bindung

Notwendige Eingaben sind sowohl das zu transformierende PIM und die Zielplattform selber – gekapselt im Platform Description Model (PDM) – sowie die Transformationsvorschriften, die im Transformation Description Model (TDM) zusammengefasst sind und die Konzepte des AAM auf die des PDM abbilden. Aufgrund seiner Sonderstellung haben wir dem TDM ein eigenes Kapitel unter dem Titel „Tranformation“ gewidmet.

Diese Klasse der PSMs ist der Endpunkt unserer Übersicht. In diesem Buch wird nicht weiter zwischen PSM und Artefakten der endgültigen Anwendung (Quellcode, Deskriptoren, …) unterschieden. Wir wollen vielmehr die letztendlich resultierende Anwendung

als sehr spezifisches PSM auffassen. Diese Betrachtungsweise wird uns im weiteren Verlauf noch Vorteile bringen, da es die Anzahl der notwendigerweise zu unterscheidenden Transformationsklassen verringert.

Nach diesem groben Überblick werden nun einige der angesprochenen Modellklassen noch einmal detaillierter betrachtet.

4.2 Computation Independent Model (CIM)

Die Klasse der *berechnungs-unabhängigen Modelle (engl. computation independent)* beschreibt das betrachtete Gesamtsystem sowie dessen *interagierender Umgebung (engl. system environment).* Dabei werden nicht nur Aspekte betrachtet, die später durch eine Anwendung unterstützt oder völlig abgelöst werden sollen, sondern Wert auf eine möglichst vollständige Beschreibung des Kontextes gelegt.

Domain Model

Eine solche Beschreibung des Gesamtsystems wird synonym auch *Domain Model (dt. Domänenmodell)* genannt. Zu seiner Modellierung wird eine Modellierungssprache verwendet, deren Notation der Terminologie und dem Verständnis der Personen entspricht, die täglich mit dem System zu tun haben. Ein Vorteil dieses Vorgehens ist, dass die *Kommunikationslücke (engl. communication gap)* zwischen Software-Entwicklern und Domänen Experten wesentlich verringert werden kann. Viele Software-Projekte erfahren ihre ersten schwerwiegenden Probleme bereits in der frühen Phase der Anforderungsanalyse. Anwender sind oft nicht in der Lage ihre Wünsche in einer Art und Weise zu formulieren, die die Entwickler verstehen. Auf der anderen Seite kann der Entwickler dem Kunden oftmals nicht verdeutlichen warum gewünschte Lösungen so (noch) nicht technisch realisierbar sind und eventuelle Alternativen verständlich aufzeigen.

Communication Gap

Tool Gap

Weiterhin unterscheiden sich die Werkzeuge und Darstellungsweisen, mit denen beispielsweise die Geschäftsprozessmodellierung durchgeführt wird, oftmals so grundlegend von denen in der Softwareentwicklung, dass bestenfalls eine manuelle teilweise Übernahme der Ergebnisse stattfindet, viele Teile auf dem Weg von der Domänenmodellierung hin zur Anwendungserstellung jedoch verloren gehen, bzw. unnötig doppelt erstellt werden. Und dies, obwohl sich die Ergebnisse prinzipiell sehr gut mit denen der Domänenmodellierung auf softwaretechnischer Seite überschneiden.

Der entscheidende Punkt ist also eine geeignete Abbildung der fachspezifischen Konzepte auf Konzepte der technischen Ebene zu finden, die von beiden Seiten verstanden und akzeptiert wird. Als gemeinsame Sprache bietet sich die UML an. Beispiele für die Abbildung eines Systems vom Domänenmodell hin zur Anwendung ohne Verlust von Arbeitsergebnissen und ohne Darstellungsbruch, werden wir im weiteren Verlauf des Kapitels und innerhalb des gesamten Buches kennen lernen.

Business Model

Stammt das modellierte System aus dem geschäftlichen Umfeld, nennt man das Domänenmodell auch *Business Model*. Die Systembeschreibungssprache eines solchen Modells enthält dann typischerweise Elemente wie: Prozess, Verantwortlicher, Prozessabhängigkeit, Konto, usw.

Requirements Model

Lässt das eben beschriebene Domänenmodell die zu erstellende Anwendung noch völlig außer Acht und stellt auch die nähere Umgebung der zu erstellenden Anwendung sowie die Prozesse dar, in die sie eingebettet werden soll, beschreibt das *Requirements Model (dt. Anforderungsmodell)* bereits dezidiert das zu erstellende Software-System bzw. die Bedingungen, die ein solches erfüllen muss, um den Anforderungen, die an ein solches gestellt werden, gerecht zu werden. Im Vergleich zum Domänenmodell bildet das Anforderungsmodell oftmals eine Untermenge desselben. Diese Untermenge wird allerdings in einem höheren Detaillierungsgrad dargestellt. Aspekte des Geschäftsprozesses, die nicht automatisiert oder maschinell unterstützt werden sollen, fallen weg und werden nicht weiter betrachtet.

Trotz der Fokussierung auf die Beschreibung des zu erstellenden Systems geschieht die Modellierung jedoch so weit wie möglich technologieneutral, indem lediglich deklarative Beschreibungen genutzt werden. Das Modell beschreibt also, *was* das zu erstellende System zu tun hat, das *Wie* bleibt weniger abstrakten Modellen in tieferen Schichten vorbehalten. Im Folgenden wollen wir uns ein Beispiel dafür anschauen, wie Systeme anwendungs-unabhängig mithilfe der UML dargestellt werden können.

Beispiel: Domänenmodell der Firma eComm

Betrachtet wird die fiktive Firma *eComm*. Die Firma will Waren über das Internet anbieten und dazu eine Website mit Shopsystem unterhalten, das Standardfunktionalitäten wie Suchen, Kontoverwaltung, Bestellen etc. umfassen soll. Zur Erstellung eines solchen Systems wird zuerst eine objektorientierte Geschäftsprozessanalyse

durchgeführt, deren Ergebnisse mittels OOGPM-Profil [OWS+03] als UML-Modelle dargestellt werden und die in der Summe das Domänenmodell des Systems bilden.

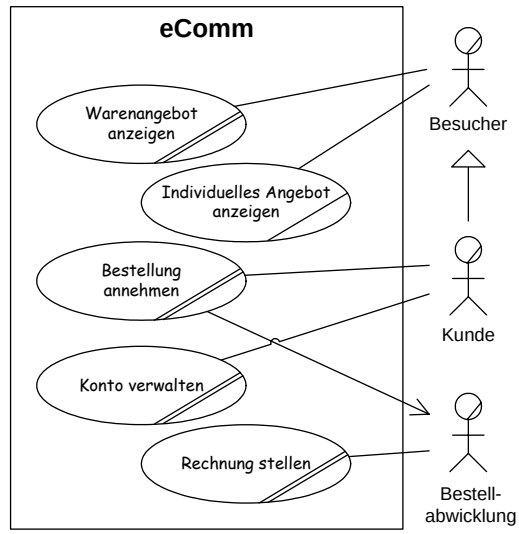
OOGPM

Die OOGPM (Objektorientierte Geschäftsprozessmodellierung) ist eine vollständige Methodik inklusive UML-Profilen die sich von der Durchführung der GP-Analyse bis hin zum Design des fertigen Software-Systems erstreckt. Anhang B fasst einige Aspekte zu diesem Thema zusammen. In diesem Kapitel wollen wir nur einen kleinen Ausschnitt aus den vorhandenen Möglichkeiten verwenden und für die Erklärung weitestgehend auf die genannte Stelle verweisen.

Besucher können im Warenangebot suchen und sich, basierend auf den zuletzt getätigten Anfragen, individuelle Angebote anzeigen lassen. Nach erfolgreicher Registrierung können sie dann als Kunde auch Waren bestellen. Eine nicht näher spezifizierte Rolle `Bestellabwicklung` nimmt die Bestellungen an und versendet zu einem geeigneten Zeitpunkt die Rechnungen an die Kunden.

Abb. 4.2 zeigt eine Sicht aus dem so entstandenen Domänenmodell, das einen Ausschnitt der gefundenen Geschäftsanwendungsfälle abbildet:

Abb. 4.2 eComm-Geschäftsanwendungsfälle (Ausschnitt)



Eine weitere Sicht auf das Domänenmodell zeigt strukturelle Aspekte des Systems, durch die Darstellung der gefundenen Geschäftsklassen in einem Klassendiagramm (Abb. 4.3).

`Besucher` erhalten auf Anfrage ein `Suchergebnis`. Dieses besteht aus einer Menge an `Angeboten`, die Preiszuordnungen zu `Waren` darstellen. Diese Art der Modellierung spiegelt den Sachverhalt wieder, dass `Ware` zu unterschiedlichen Zeitpunkten unterschiedliche Preise besitzen kann. Erfolgt eine Bestellung, setzt sich diese aus den ausgewählten `Angeboten` zusammen. Bevor die `Bestellung` bestätigt wird, wird zuerst im `Lager` nachgefragt, ob die `Ware` in ausreichender Stückzahl vorhanden ist. Die fällig werdende Zahlung kann durch Bankeinzug oder per Kreditkarte erfolgen.

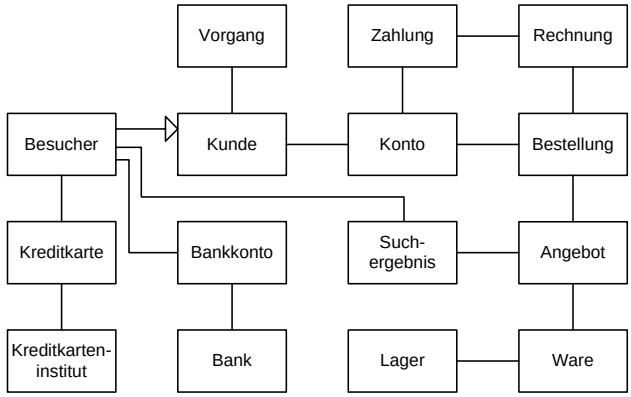


Abb. 4.3 Domänenmodell eComm

Abschließend zu diesem Abschnitt noch eine Sicht auf den Verhaltensaspekt des Modells: Abb. 4.4 zeigt den prinzipiellen Ablauf des Geschäftsvorfalles `Bestellung annehmen`.

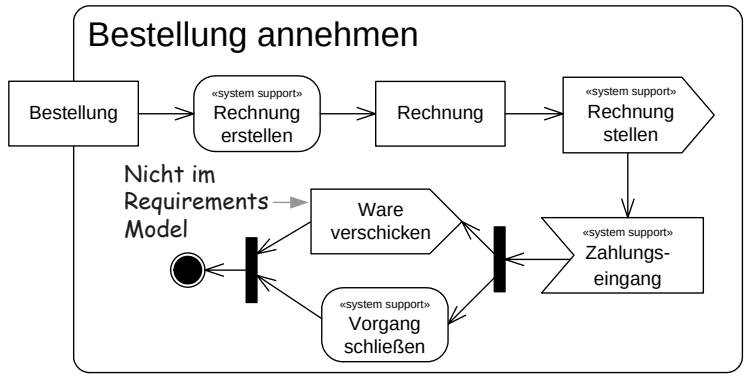


Abb. 4.4 Ablauf GA Bestellung annehmen

Eine `Bestellung` wird entgegengenommen und daraus die `Rechnung` erstellt. Diese wird zur `Zahlung` an den Kunden

weitergeleitet, nach Zahlungseingang die Ware verschickt und damit der vereinfachte Vorgang abgeschlossen.

Kennzeichnung von Elementen des Requirement Models

Ebenfalls in der Abbildung dargestellt ist eine Möglichkeit die Unterscheidung zwischen Domänen- und spezifischerem Requirements Model zu unterscheiden, ohne zwei Sichten erstellen zu müssen: Elemente, die durch die zu erstellende Anwendung unterstützt oder automatisiert werden sollen, werden durch den Stereotypen `«system support»` [Oes05] gekennzeichnet. Für den Übergang in die nächste Abstraktionsebene bedeutet dies z. B., dass das Element selbst, bzw. ein geeignetes transformiertes Äquivalent im Folgemodell auftauchen muss.

4.3 Plattform Independent Model (PIM)

Bevor wir uns dem zentralen MDA-Modell zur Beschreibung der zu entwickelnden Anwendung – dem PIM – widmen können, müssen wir uns zuerst mit weiteren Konzepten vertrauter machen, die in der Welt des PIMs eine prominente Rolle spielen:

- Mit dem Begriff der *Plattform*, da die Aussage, ein System sei plattform-unabhängig erst dann Sinn macht, wenn man angibt bezüglich welcher Plattform die Aussage gilt,
- der Menge der *Aspekte* (eines Software-Systems bzw. die einer Familie von Software-Systemen), die sich mittels PIMs darstellen und sich in abgrenzbare, kohärente Gebiete zergliedern lassen, die Domänen genannt werden,
- sowie dem Begriff der *Architektur von PIMs*, die durch das Architekturmetamodell (AMM) vorgegeben wird. Sie spezifiziert das Aussehen gültiger Modelle dieser Klasse durch die Angabe der Konstrukte und entsprechender Vorgaben zu deren Anwendung und definiert so, wie die Modellierung von Aspekten einer Domäne zu geschehen hat.

Plattformbegriff

Beginnen wir damit den Begriff 2Plattform“ für uns ein wenig handfester zu machen. Halten wir uns auch hierbei eng an die Spezifikation der OMG, definiert sich der Begriff wie folgt [MDAGuide]:

Definition der OMG

> *„Eine Plattform ist eine kohärente Menge von Subsystemen und Technologien, die auf dieser Plattform lauffähige Anwendungen benutzen können. Die Benutzung geschieht durch die Anwendung genau spezifizierter Schnittstellen und Gebrauchsmustern, ohne*

Kenntnis darüber vorauszusetzen, wie die angebotene Funktionalität technisch realisiert wird.“

Wir sehen diesen Sachverhalt in Abb. 4.5 verdeutlicht. In diesem (vereinfachten) Schichtenmodell sehen wir zuunterst die Ebene, auf der letztendlich die Ausführung unserer Anwendungen stattfindet. Diese Schicht kann, je nach Sicht, sowohl aus Hardwarekomponenten bestehen, jedoch genauso gut eine weitere Plattform repräsentieren, von der abstrahiert wird. Das Prinzip der Fraktalität greift also auch hier und zeigt sich in der „Stapelbarkeit“ von Plattformen. Eine Plattform ist die Ausführungsschicht der Plattform auf der nächsthöheren Abstraktionsebene. Ein Gedanke, der sich in der MDA im Konzept der PIMs und PSMs wiederfindet.

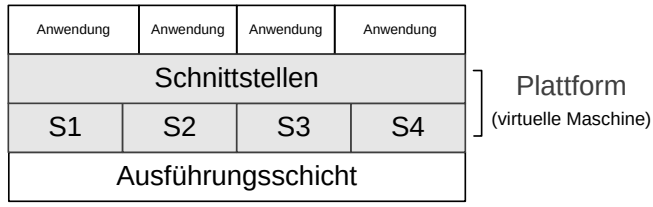


Abb. 4.5 Plattform und virtuelle Maschine

Über diese Schicht kommen wir zu Subsystemen, die auf die Funktionalität der Ausführungsschicht zugreifen (Sx), und diese Funktionalität durch eine Fassade [GOF] in Form von Schnittstellen den Anwendungen der obersten Schicht zur Verfügung stellen. Die Zusammenfassung dieser Systeme und ihrer Fassade(n) nennt man *Plattform* oder *virtuelle Maschine*. Der Begriff der virtuellen Maschine enthält in vielen Definitionen die zusätzliche Betonung der Kapselung zwischen den einzelnen Anwendungen innerhalb der Plattform, welche aber für unsere Betrachtung nicht näher von Bedeutung ist. Wer trotzdem tiefer in die Welt der virtuellen Maschinen eintauchen will, sei auf eine Einführung in [VM] verwiesen.

Arten von Plattformen

In der MDA-Spezifikation definiert die OMG weiterhin drei Arten von Plattformen, deren Unterscheidung in der Welt der MDA eine Rolle spielt (vgl. [MDAGuide]):

- *generisch*: umfasst logische Konzepte, die technologieneutralen Charakter haben. Beispiele: Konzept der Objektorientierten Programmierung (OOP), Komponentenbasierte Softwareentwicklung (Component Based Development, CBD)
- *technologiespezifisch*: Umsetzung von Konzepten innerhalb einer Technologie bzw. eines Standards. Beispiele: Common

Object Request Broker Architecture (CORBA), Java Enterprise Edition (Java EE), .NET

- *herstellergebunden* (proprietär): Umsetzung technologiespezifischer Konzepte durch einen Hersteller. Beispiele: Microsoft .NET, Mono (.NET), IBM WebSphere (Java EE), BEA WebLogic (Java EE), Borland VisiBroker (CORBA), IONA Orbix (CORBA), …

PIMs sind technologie- und herstellerneutral

Das plattform-unabhängige Modell (PIM) der MDA bezieht sich auf die Unabhängigkeit von technologiespezifischen sowie herstellergebundenen Plattformen. Dabei ist eine Systembeschreibung notwendigerweise niemals völlig plattform-unabhängig. So werden PIMs in der MDA mittels UML-Konstrukten modelliert und stützen sich so bereits implizit auf generische Plattformen, wie sie bereits die UML selbst bildet, oder transitiv auf das der UML unterliegende Konzept der Objektorientierung, ab. Im Falle dieser generischen Plattform-Abhängigkeiten ist aber deutlich hervorzuheben, dass diese keine Einwirkung auf die spätere, tatsächliche Realisierung des modellierten Systems haben müssen. So kann eine Anwendung mit objektorientierten Konzepten modelliert, aber durch Anwendungen in prozeduraler Sprache (wie beispielsweise in der Programmiersprache C) realisiert werden.

*Plattform-Abhängigkeit ist **kein** a priori Konzept*

Die Unterscheidung von Modellen in plattform-abhängig und plattform-spezifisch (in Kürze vorgestellt) folgt somit keinem „Schwarz-Weiß-Raster“, in das sich Modelle *a priori* pressen lassen. Es handelt sich vielmehr um relative Konzepte, die lediglich die Aussage zulassen, dass einige Modelle „im Grunde“ plattform-spezifischer sind als andere. Erst die genaue Angabe, was in der konkreten Situation unter den Begriff der Plattform gefasst werden soll, macht die eindeutige Zuordnung von Modellen zu einer der beiden Klassen möglich und sinnvoll. In unserem Falle wurde diese Klarheit durch die Definition einer Plattform als „technische Basis“ auf der eine Anwendung läuft geschaffen.

Domänenbegriff

Im vorigen Abschnitt haben wir gesehen, das PIMs den Fokus auf die Modellierung von Systemaspekten legen, die so weit wie möglich technologie- und herstellerneutral abgebildet werden. Im Folgenden werden wir sehen, welche konkreten Aspekte ein PIM nun tatsächlich darstellen kann und ferner, wie sich diese weiter in klar abgrenzbare, kohärente Gebiete zergliedern lassen.

Betrachtung von Systemfamilien

Die Abgrenzung dieser Gebiete geschieht durch die Ausweitung der Betrachtung weg vom Software-Einzelsystem hin zur Untersuchung der Gemeinsamkeiten und (der klar definierten) Mengen der spezifischen Unterschiede ganzer Familien von Software-Systemen. Diese werden geordnet, zusammengefasst und gruppiert und schließlich als Domänen bezeichnet.

Vertikale und horizontale Domänen

Zur weiteren Unterscheidung von Domänenarten haben sich die Begriffe vertikale bzw. horizontale Domäne eingebürgert, die wie folgt definiert sind:

- Eine horizontale Domäne beschreibt einen – zu meist technischen – Querschnittsaspekt eines Systems, also z. B. Sicherheit, Transaktionalität, Persistenz sowie *das* Standardbeispiel dieser Aspektart: Logging.
- Vertikale Domänen hingegen umfassen die Konzepte verschiedener fachlicher Anwendungsgebiete und unterteilen so Anwendungen in fachliche Gruppen wie beispielsweise Versicherung, Gesundheit oder Versorgung.

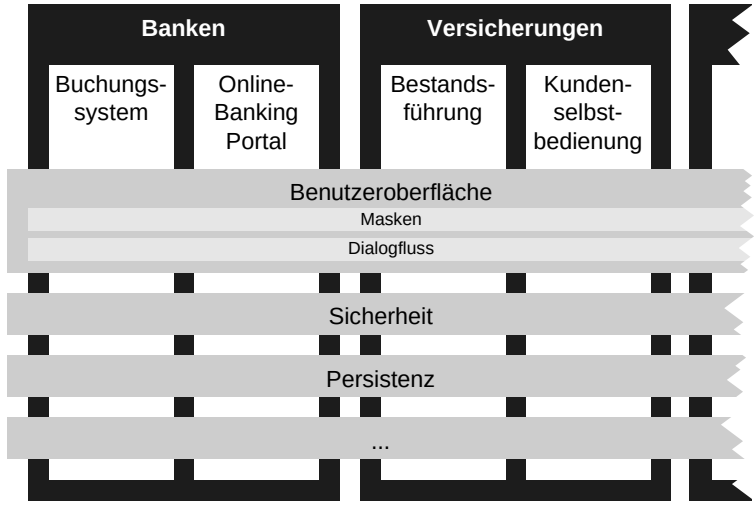


Abb. 4.6 Horizontale und vertikale Domänen

Weiterhin kann eine Domäne in Subdomänen unterteilt werden, die wiederum in Subdomänen untergliedert werden können und so weiter und so fort. Sprich: Auch Domänen sind fraktal definiert.
So kann die vertikale Domäne Versicherung z. B. weiter unterteilt werden in die Domänen „Bestandsführung“ und „Kundenselbstbedienung“ (siehe Abb. 4.6), während sich die horizontale Domäne

„Benutzeroberfläche“ beispielsweise in die Domänen „Masken“ und „Dialogfluss“ untergliedern ließe.

Wollen wir die unter Abschnitt 4.2 betrachtete Domäne in dieses Schema einordnen, fällt sie unter die Gruppe der vertikalen Domänen (in dem dort dargestellten Beispiel ist dies konkret die Domäne „Handel“). Unser weiteres Interesse soll aber nun den horizontalen Domänen gelten.

4.4 Architecture Metamodel (AMM)

Nachdem wir nun wissen, was mithilfe eines PIMs dargestellt werden kann, nämlich technologieneutrale Aspekte horizontaler Domänen von Anwendungen, die in vertikale Domänenklassen eingeordnet werden können, ist es Zeit sich zu fragen, wo die Regeln zur Erstellung eines PIM festgehalten werden, also nach dem Wie der plattform-unabhängigen Modellierung zu forschen.

Architektur

Rufen wir uns dazu zunächst noch einmal die Definition des Begriffs *Architektur* aus Kapitel 2 in den Sinn:

> *„In der Architektur finden sich die Organisation der Teile eines Systems, deren Verhalten sowie die Beziehungen unter den einzelnen Komponenten wieder. Beziehungen können zudem an Bedingungen geknüpft sein, die zur Erreichung des Systemzwecks erfüllt werden müssen“*

Das Metamodell einer Architektur (ein Architektur Metamodell) beschreibt also die wesentlichen Konzepte, Abstraktionen, Schnittstellen und Muster, die zur Modellierung eines Systems angewendet werden dürfen.

Im Falle der MDA geschieht die wesentliche Modellierung, die als Ausgangspunkt und Wissensspeicher dient, auf einer plattformunabhängigen Ebene, innerhalb der Klasse der PIMs. Das AMM liefert die dazu notwendigen Modellierungssprachen zur Darstellung der einzelnen Aspekte oder Aspektklassen, die wir an voriger Stelle schon einmal unter den Begriff der Domänen gefasst haben. Somit sammelt das AMM die zur plattform-unabhängigen Beschreibung des Systems notwendigen Domänen-Spezifischen Sprachen (DSLs).

Im Folgenden sollen einige Beispiele solcher DSLs in Form von UML-Profilen gezeigt werden, um das eben Dargestellte mit Leben zu füllen. Im letzten Abschnitt innerhalb des Themenkomplexes soll die Beispielanwendung, die wir bereits aus dem Abschnitt über die CIMs kennen, mithilfe dieser Profile weiter mit Informationen

angereichert werden und so das Zusammenspiel mehrerer Profile gezeigt werden.

Beispiel: DSL der Domäne „Sicherheit“

In Abb. 4.7 ist der Ausschnitt einer solchen Sprache in Form eines einfachen UML Profils für die Domäne „Sicherheit“ dargestellt. Beispielhaft erläutert bedeuten die dort definierten Sprachkonstrukte in Form von Stereotypen das Folgende:

- Der Stereotyp `«AccessPermission»` kann auf eine Assoziation angewendet werden und bedeutet, dass eine Zugriffsberechtigung besteht, deren Art über das Attribut `p` an dem Stereotypen genauer spezifiziert wird.
- `«ExcutionPermission»` ist ein Stereotyp der die Erlaubnis zum Ausführen einer Operation festlegt. Über das Attribut `actor` wird bestimmt welche Rollen/Akteure dieses Recht innehaben.
- Wird der Stereotyp `«Trusted»` auf eine Instanz der Metaklasse `CommunicationPath` angewendet, so heißt das, dass sichergestellt werden muss, dass der entsprechende Kommunikationspfad vertrauenswürdig ist.

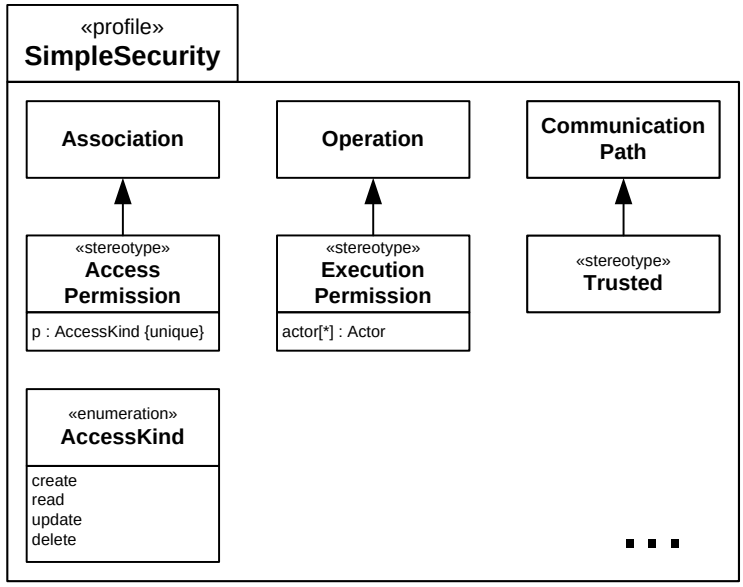


Abb. 4.7 SimpleSecurity-Profil

Mit diesem Profil ausgestattet können wir die Sicherheitsaspekte unserer Anwendungen festlegen, ohne uns auf eine konkrete Tech-

nologie zur Umsetzung der geforderten Maßnahmen festlegen zu müssen.

Sicherheit ist aber nur ein Aspekt, der bei der Erstellung unserer exemplarischen eCommerce-Anwendung bedacht und von unserer Architektur abgedeckt werden muss. Weitere Profile in diesem Sinne, von denen wir einige im weiteren Verlauf noch kennen lernen werden sind:

- `SimpleWebLayout` mit der wir die Darstellung der benötigten Web-Seiten spezifizieren können,
- `SimpleWebFlow` mit dem der Dialogfluss der Anwendung beschrieben wird,
- `SimplePersistence`, welches für die Spezifikation der Datenhaltung herangezogen wird,
- `SimpleDDD`, zur Darstellung der Konzepte des Domain-Driven Designs im Sinne des von Eric Evans in [Eva03] beschriebenen, gleichnamigen Ansatzes,
- etc.

Bündelung von Profilen zur Abdeckung vertikaler Domänen

All diese Profile ermöglichen die Darstellung von Aspekten jeweils einer Domäne, die wir zuvor bereits der Klasse der horizontalen Domänen zugeordnet haben. Da bei der Abdeckung kompletter vertikaler Domänen (den Anwendungsarten) oft mehr als ein horizontaler Aspekt eine Rolle spielt und somit auch in einem vollständigen Systemmodell auftauchen muss liegt es nahe, geeignete Untermengen der verfügbaren Profile zu einer Profilsammlung zu bündeln, die die gesamte Bandbreite der darzustellenden vertikalen Aspekte abdeckt. Stehen erst einmal mehrerer dieser „Profilbündel“ zur Verfügung, müssen zur Modellierung eines Systems nicht mehr alle Unterprofile mühsam zusammengesucht, sondern nur noch eine Profilsammlung nach der Art des zu erstellenden Softwaresystems ausgewählt und die für die Anwendung relevanten Aspekte modellhaft dargestellt werden.

Wollten wir mit diesem Ansatz beispielsweise ein Gesamtprofil für die Spezifikation von eCommerce-Anwendungen zusammenfassen, könnte dies wie in Abb. 4.7 aussehen.

Bevor wir unser Beispiel fortführen und einige Sichten der PIM Ebene auf das zu erstellende Softwaresystem vorstellen, wollen wir noch weitere Beispiele für DSLs in Form von Profilen vorstellen, darunter im Folgenden eine Art (Abb. 4.9), die sich in der Art der

abgedeckten Domäne wesentlich von den bisher vorgestellten Beispielen unterscheidet.

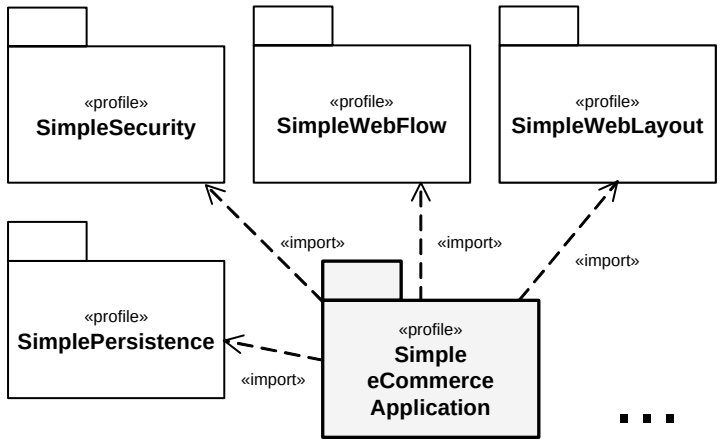


Abb. 4.8 Kombination von Profilen

Beispiel: DSL zur Modellierung funktionaler Variabilität

Betrachten wir die bisher vorgestellten Profile noch einmal unter dem Aspekt der Gemeinsamkeiten und Variabilitäten, lagen die Variabilitäten bisher allesamt im technischen Bereich. So haben beispielsweise alle Anwendungen, die das Persistenzprofil anwenden die Gemeinsamkeit des Bedürfnisses nach persistenter Datenhaltung. Die Variabilität liegt dann in der Auswahl einer konkreten Technologie (wie Java EE oder Hibernate), die erst in der nächsten Abstraktionsstufe und Modellklasse (den PSMs) aufgelöst wird. Dort muss man sich letztendlich für eine der angebotenen technischen Realisierungen entscheiden, um zu einer lauffähigen Anwendung zu gelangen.

Modellierung funktionaler Variabilität mit UML

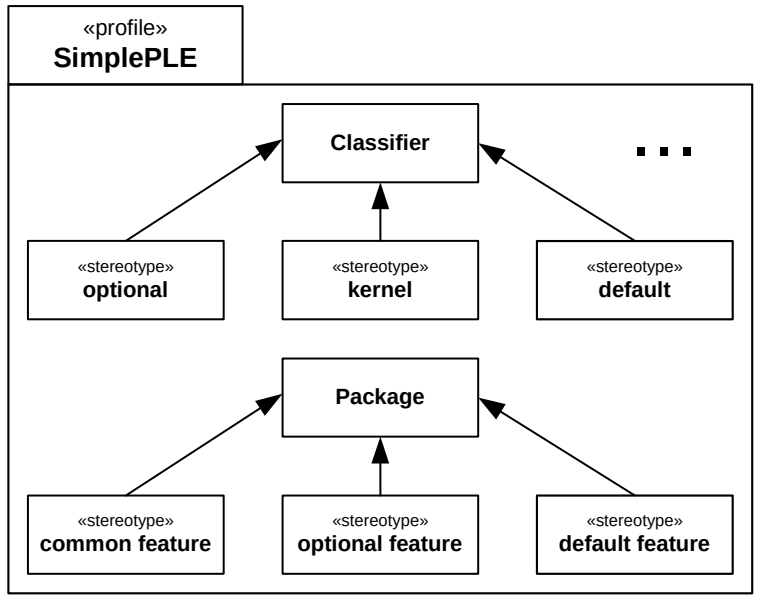
Mit den Mitteln der UML ist jedoch auch die Modellierung funktionaler Variabilitäten möglich. Der grundsätzliche Gedanke stammt aus dem Bereich der *Software-Produktlinien (engl. software product lines)* [CN01], die Familien von (Software-)Systemen repräsentieren, die einen Kern gemeinsamer Funktionalitäten besitzen, sich jedoch durch zusätzliche optionale oder alternative Details der umgesetzten Features unterscheiden.

Das Profil `SimplePLE` ist [Gom04] entlehnt und bietet die Möglichkeit folgende Variabilitäten der im Produkt enthaltenen Funktionalität darzustellen:

- Funktionen, die in allen Mitgliedern der Produktlinie vorhanden sind (im Modell mit dem Stereotyp «kernel» gekennzeichnet),
- Funktionen, die optional zur Grundfunktionalität hinzugenommen werden können (Stereotyp «optional»),
- und solche, die Alternativen darstellen, also Optionen, die sich gegenseitig ausschließen (hier nicht gezeigt, Stereotyp «variant»)

Der Stereotyp «default» zeigt an, dass es sich um eine optionale Funktionalität handelt, die als Standard angenommen wird, wenn keine explizite Auswahl erfolgt. Funktionalität selbst wird durch die Darstellung der identifizierten Geschäftsanwendungsfälle als Use Cases abgebildet, die Stereotypen selbst können jedoch bereits auf Classifier angewendet werden, was sie genereller verwertbar macht (wir werden in Kürze ein Beispiel für die Verwendung des Profils sehen).

Abb. 4.9
SimplePLE Profil



Features (im Jargon der Software-Produktlinien in Software gegossene, funktionale Anforderungen) können unter einem Sammelbegriff zusammengefasst und als Sammlung einer Menge zusammengehöriger Use Cases in Paketen dargestellt werden. Die zur Verfügung stehenden Stereotype zeigen an, ob es sich um Kernfeatures, optionale Features oder Alternativen handelt. Die Bedeutung von

Kern-, optional und alternativ entspricht dabei der, die schon für Anwendungsfälle besprochen wurde.

Beispiel: DSL Domain-Driven Design™

Als letztes Beispiel wollen wir noch ein Profil vorstellen, das eine Modellierung der Konzepte ermöglicht, die Eric Evans in seinem Buch *Domain-Driven Design* vorschlägt [Eva03] und die folgenden drei Stereotype enthält:

- *Entitäten* (`«entity»`) sind Klassen, deren Instanzen persistenten Charakter haben. Sie besitzen eine Identität sowie einen eigenen Lebenszyklus.
- *Value-Klassen* (`«value»`) beschreiben Entitäten, besitzen selbst aber keine – für den Anwendungszweck relevante – Identität. Typischerweise übernehmen Entitäten (`«entity»`) die Aufgabe der Verwaltung dieser Klassen (Zugriffssteuerung, Persistenz, …). Sie dienen oft zum Austausch von Daten zwischen Schichten oder Komponenten und stellen damit so etwas wie Transportobjekte dar.
- *Service-Klassen* (`«service»`) kapseln zustandslose Funktionalität, die nicht den übrigen Elementen des Domänenmodells zugeordnet werden kann, ohne das Modell zu 'verwässern' bzw. das Separation-of-Concern Prinzip zu verletzen. In unserem Beispiel stellen sie Fassaden zu externen Systemen dar, gehören aber trotzdem zum PIM, da die konkrete Kommunikation mit dem System nicht vorgegeben wird.

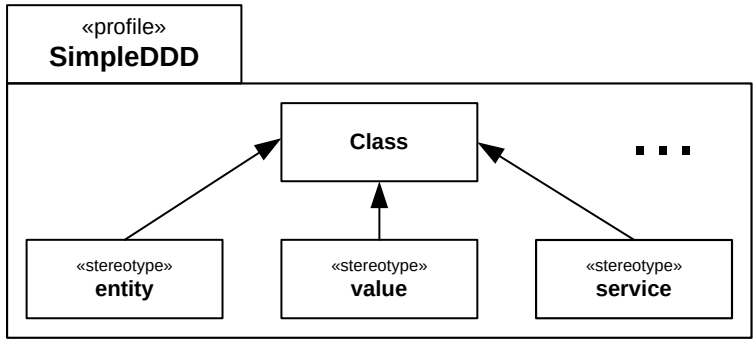


Abb. 4.10 SimpleDDD Profil

Abb. 4.10 zeigt das Profil in der UML-Darstellung.

Beispiel: All together now – PIM

Schauen wir nun, wie einige der vorgestellten Profile zur Modellierung der in Abschnitt 4.1 vorgestellten Beispielanwendung auf PIM-Ebene verwendet werden können.

Abb. 4.11 Auszug Use Case Modell

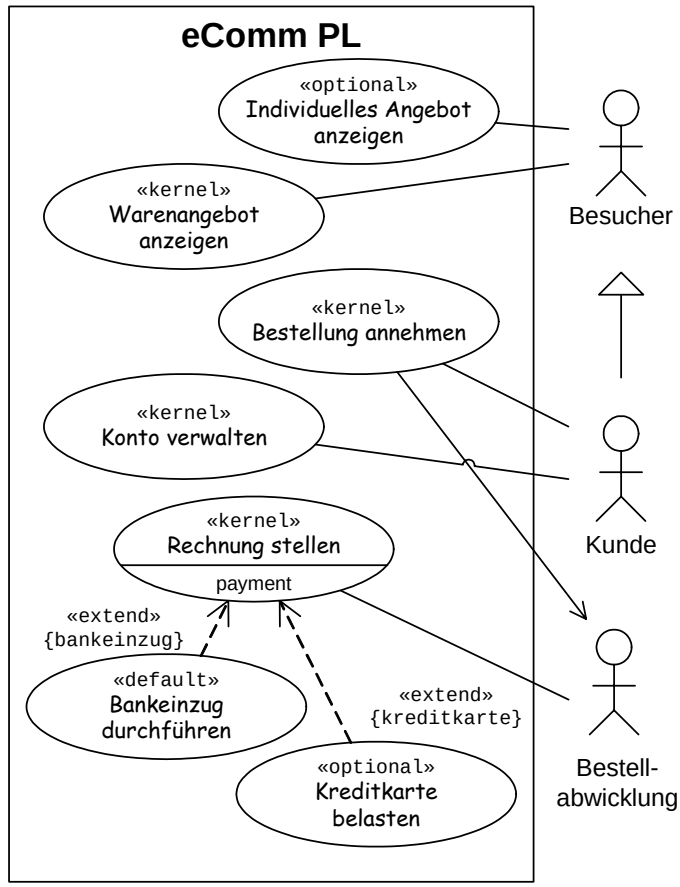


Abb. 4.11 zeigt eine Variation des Use Case Modells, das wir bereits aus dem CIM kennen. In der Strategie der betrachteten Beispielfirma gibt es eine Neuerung zu vermelden: Das Management von *eComm* hat sich dazu entschieden das zu entwickelnde Shopsystem, das ursprünglich nur zum Vertrieb der eigenen Produkte gedacht war, und dessen Funktionalität bereits durch das Anforderungsmodell festgelegt wurde, nicht für sich selbst zu behalten, sondern gleich mit in den Warenkorb aufzunehmen und ebenfalls als Produkt zu vermarkten.

Um für unterschiedliche Märkte und Kundenwünsche gerüstet zu sein, entschließen sich die Verantwortlichen zur Beauftragung der Entwicklung einer Produktlinie, anstatt sich auf ein Einzelsystem zu beschränken. In unserem Beispiel sind zwei Features dargestellt, die vom gemeinsamen Kern der Produktfamilie abweichen: die Darstellung eines individuellen Angebotes, basierend auf bereits getätigten Suchanfragen oder Bestellungen sowie die Bezahlmöglichkeit mit Kreditkarte.

Während die Anzeige von Individualangeboten eine rein optionale Funktionalität darstellt, die ein möglicher Käufer des Shopsystems hinzukaufen kann, ist mit der Auswahl der Bezahlmöglichkeiten noch eine Besonderheit verknüpft. Standard ist die Bezahlung mittels Bankeinzug, es besteht jedoch auch die Möglichkeit der Bezahlung mittels Kreditkarte. Welche Zahlungsweise zum Zuge kommt und in der lauffähigen Anwendung enthalten ist, entscheidet die Auswertung der *Constraints* an den Extend-Beziehungen.

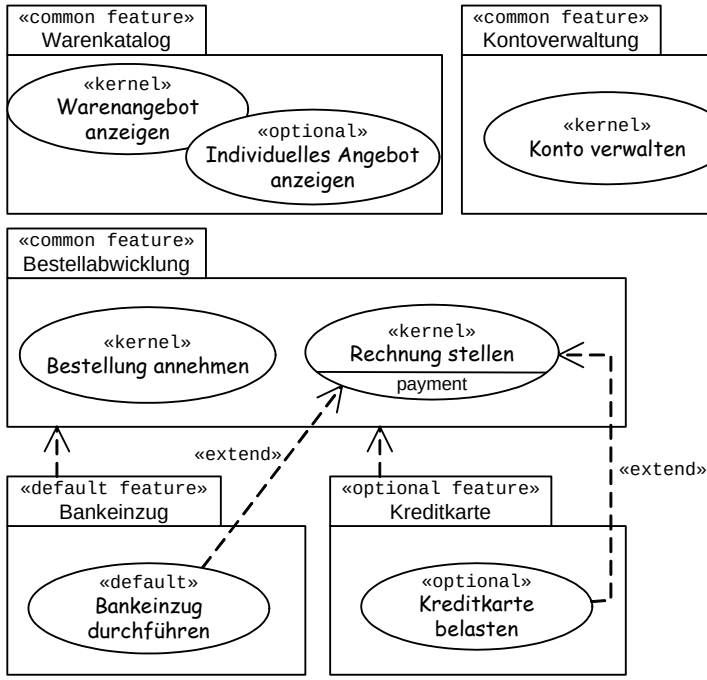


Abb. 4.12 Zuordnung von Anwendungsfällen zu Features

Enthält ein Mitglied der Systemfamilie beide Optionen, entscheidet sich die Auswahl einer konkreten Vorgehensweise (Bezahlung per Bankeinzug oder Belasten der Kreditkarte) erst zur Laufzeit des Systems. Dies geschieht in der Regel durch den Kunden selbst.

Verlangt der Käufer des Systems lediglich die Möglichkeit der Bezahlung per Kreditkarte, können in einem Modell der nächstspezifischeren Abstraktionsklasse die Constraints `bankeinzug` auf `false` sowie `kreditkarte` auf `true` gesetzt werden.

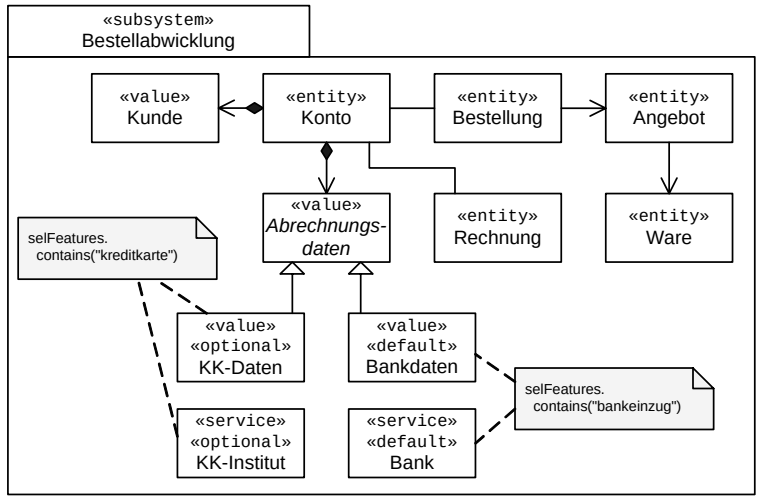
Abb. 4.12 zeigt die Zuordnung der angesprochenen Anwendungsfälle zu konkreten Produktfeatures des *eComm* Shopsystems.

In unserem Beispiel sind die Anwendungsfälle `Bestellung annehmen` und `Rechnung stellen` unter dem Feature `Bestellabwicklung` zusammengefasst.

Als weiteres Ergebnis der Bemühungen entsteht das Modell einer ersten Architektur zur Realisierung dieses Features. Abb. 4.13 zeigt das Subsystem, das später für den Bereich der `Bestellabwicklung` verantwortlich sein soll. Es knüpft an das zuvor betrachtete Klassendiagramm des CIMs an, ist jedoch auf die reine Erfüllung der Anforderungen zurechtgestutzt worden.

Das Modell enthält bereits erste Informationen über die Persistenz von Objekten sowie über die Anbindung externer Systeme. Zur Darstellung der gewünschten Aspekte wurden dabei die Profile `SimplePLE` und `SimpleDDD` kombiniert.

Abb. 4.13 Plattformunabhängiger Design-Prototyp



`Konto` enthält eine abstrakte Value-Klasse des Typs `Abrechnungsdaten`. Abhängig von den ausgewählten Features eines Produktfamilienmitglieds wird polymorph die Klasse `KK-Daten`, bzw. `Bankdaten` in die Anwendungsarchitektur mit aufgenommen.

Ob ein Feature zur Realisierung ausgewählt wurde oder nicht, wird in `selFeatures` festgehalten. Eine Auswertung findet mittels OCL statt: Ist das Feature `kreditkarte` in der Menge der gewählten Funktionen enthalten, wird die Value-Klasse `KK-Daten` integriert. Entsprechendes gilt für die Klasse `Bankdaten` sowie für die zugehörigen Service-Klassen.

Auch Sicherheitsaspekte können bereits an dieser Stelle technologieunabhängig modelliert werden. Abb. 4.14 zeigt die Beschränkungen, die bei Zugriffen auf die Entität Konto Gültigkeit besitzen.

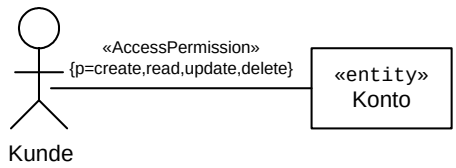


Abb. 4.14 SimpleSecurity angewendet

Dargestellt mit Elementen des SimpleSecurity Profils (siehe oben) wurde folgender Sachverhalt festgehalten: Akteure des Typs `Kunde` haben Zugriff auf Entitäten des Typs `Konto`. Der Zugriff ist dabei nicht beschränkt, da Kunden sowohl Kontodaten *anlegen*, *lesen*, *verändern* sowie *löschen* dürfen.

4.5 Platform Description Model (PDM)

In unserem Überblick über den Modelllebenszyklus (Abb. 4.1) nähern wir uns nun dem zweiten großen Abstraktionsstufen-Übergang: dem Wechsel zwischen plattform-unabhängiger und plattform-abhängiger Modellierung. Hier werden die bisher technologieneutral betrachteten Architekturkonzepte mit domänenspezifischen Aspekten konkreter Realisierungstechnologien angereichert (man spricht dabei auch von einer Technologie-Projektion bzw. Technologie-Bindung), um so dem Endziel des Vorhabens, einer ausführbaren Anwendung, ein weiteres Stück näher zu kommen.

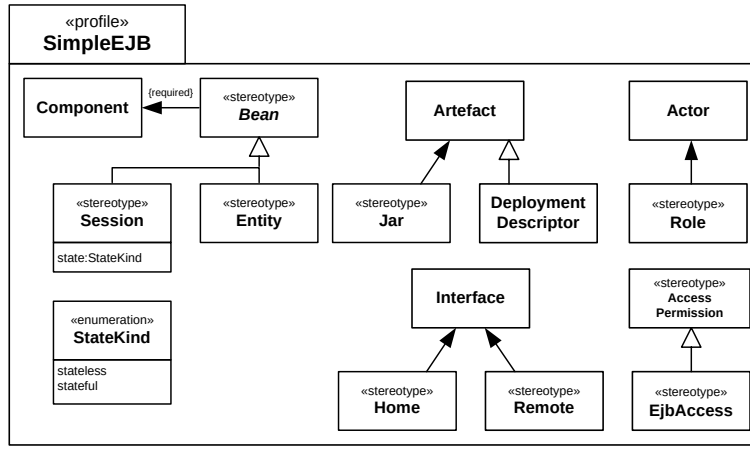
Die zum Einsatz kommenden Technologien, bzw. logische Technologiegruppen werden dabei ebenso einzeln als Plattform bezeichnet, wie auch die Menge aller verwendeten Technologien zusammen als die Plattform bezeichnet wird, auf der letztlich die gesamte Anwendung ausgeführt wird.

Ähnlich wie beim Übergang zwischen CIM und PIM, müssen auch hier die Regeln festgehalten werden, die eine PSM-Architektur bei Darstellung der gewählten Aspekte erfüllen muss, um als valide zu gelten. Wir brauchen also ein Metamodell, das die Regeln zur Modellierung festhält, und wollen dieses im Folgenden unter dem Begriff des Platform Description Model (PDM) (dt. etwa plattformbeschreibendes Modell) näher untersuchen. Die Darstellung des PDM geschieht im Wesentlichen mittels der, schon aus dem AMM bekannten, UML Profile, die das UML-Metamodell um die Aspekte erweitern, die zur Darstellung der technischen Domänen benötigt werden, bzw. durch die Schaffung eines gänzlich neuen Metamodells auf MOF-Basis.

Beispiel: DSL Java Enterprise Edition (Java EE)

Abb. 4.15 zeigt die vereinfachte Darstellung eines Java EE EJB Profils in UML 2.x, das uns in ähnlicher Form bereits im Kapitel über Modellierung (Kapitel 3) begegnet ist. Daher wollen wir auf Einzelheiten der Bedeutung der Profilelemente nicht näher eingehen.

Abb. 4.15 Einfaches EJB-Profil



Neu sind lediglich die zusätzlichen Definitionen der Elemente `EjbAccess`, einem Stereotypen, der den bereits aus dem Simple-Security Profil bekannten Stereotypen `«AccessPermission»` weiter spezialisiert sowie `«Role»`, einer Erweiterung des Akteur-Elementes aus der UML Superstructure. Zusätzlich wird das Artefakt des `DeploymentDescriptors` eingeführt.

Als Erweiterung der Superstructure besitzt SimpleEJB direkten Zugriff auf alle dort definierten oder importierten Elemente. Die

Voraussetzung zur Benutzung von `AccessPermission` wird durch den Import von SimpleSecurity gewährleistet (Abb. 4.16).

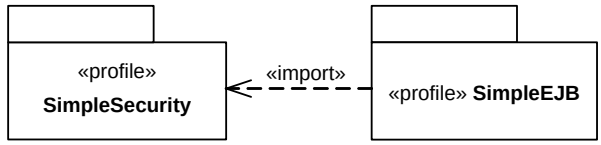


Abb. 4.16 Import von SimpleSecurity in SimpleEJB

4.6 Platform Specific Model (PSM)

Aufbauend auf dem gerade beschriebenen PDM und damit noch näher an der letztendlich resultierenden Applikation angesiedelt, ist das *plattform-spezifische Modell (engl. Platform Specific Model) (PSM).* Es berücksichtigt, anders als die in den vorigen Abschnitten vorgestellten Modelle, bereits die Plattform, die der Anwendung zur Laufzeit unterliegt, und mit deren zur Verfügung gestellten Technologien die Anforderungen des Systems erfüllt werden sollen. Dazu zählen hauptsächlich Systemaspekte, die die benutzte Middleware betreffen, die zur Realisierung der spezifizierten Applikation benötigt wird.

Die Erstellung eines PSMs erfolgt unter Nutzung des PDMs und der dort spezifizierten DSLs, die in vielfältigen Formen vorliegen können. So gehören zu einem PSM eine Vielzahl von Aspektbeschreibungen, die durch UML-Modelle, resultierend aus einem UML-Profil, oder MOF-Metamodell, Quellcodestücken (konsistent zur Plattform), aber auch durch Konfigurations- und Deploymentwissen aufbereitet in geeigneter Form, beschrieben sein können.

Bevor wir unsere Beispielanwendung mit den Mitteln des PDMs und TDMs der Technologie-Bindung und damit dem Übergang in die Modellklasse der PSMs unterziehen, wollen wir uns noch mit einer besonderen Unterart der PSMs befassen: dem Quellcode der Anwendung.

Auch Quellcode ist ein PSM

In diesem Buch wollen wir Quellcode ebenfalls der Klasse der PSMs zuordnen und ihn damit in die Familie der MDA-Modellarten einreihen. Wie wir bereits im Kapitel über Modellierung gesehen haben, ist dies kein Widerspruch, sondern nur eine konsequente Fortführung des Gedankens, ein Modell immer weiter mit Zusatzinformationen anzureichern, bis alle Details, die zur Ausführung des Modells als Anwendung benötigt werden, zur Verfügung stehen.

Dieser Punkt ist mit dem Vorliegen des vollständigen Quellcodes (inklusive aller benötigten weiteren Artefakte wie Deskriptoren, usw., die wir an dieser Stelle vereinfachend unterschlagen wollen) erreicht und stellt quasi den vorläufigen Endpunkt des betrachteten Modell-Lebenszyklus dar.

Neben der theoretischen hat eine solche Betrachtungsweise auch eine konkret-praktische Bedeutung: Im nächsten Kapitel werden wir sehen, dass durch die Einordnung von Quellcode in die Klasse der PSMs, die Unterscheidung zwischen Transformationen zwischen Modellen (*Model-to-Model*) und der „klassischen" Codegenerierung (*Model-to-Text*) entfällt. Dies erlaubt eine einheitliche Behandlung der Transformationen und hat weitere Vorteile in der Definition von Regeln zur Codegenerierung, auf die wir an dieser Stelle jedoch nicht vorgreifen wollen und daher mit unserem Beispiel fortfahren:

Beispiel: All together now II – PSM

Ist unser Beispielsystem bis jetzt ohne Darstellung expliziter Realisierungstechnologien modelliert, kommen nun also genau diese Informationen zum Modell hinzu.

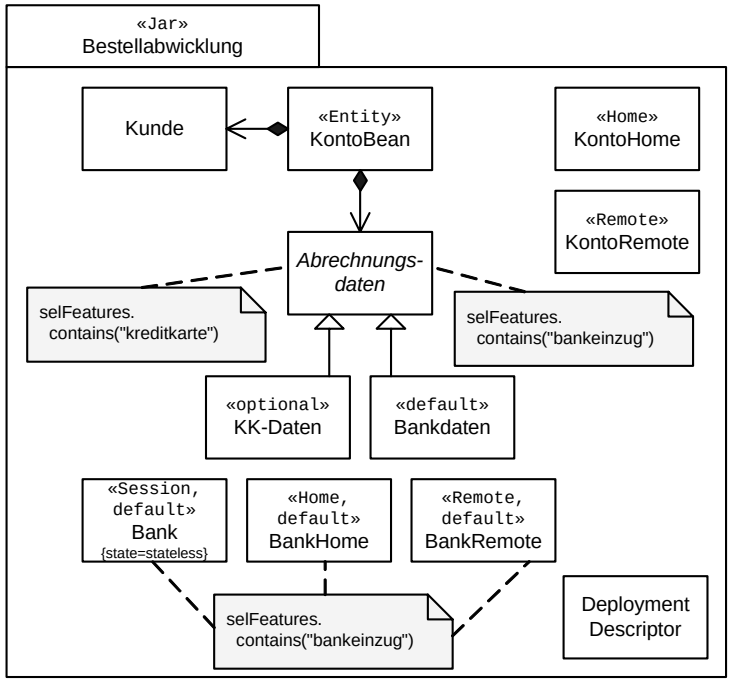


Abb. 4.17 Unser Beispiel als PSM (Ausschnitt)

Entscheiden wir uns beispielsweise dazu EJB als unterliegendes Framework zu nutzen, ergibt sich das Modell in Abb. 4.17.

«entity»s werden zu «EntityBean»s, «service»s zu «SessionBean»s. Zusätzlich werden Home- und Remote-Interfaces generiert. Die Markierung der Elemente als «default», «optional» usw. bleibt erhalten. Zusätzlich wurde die Entscheidung getroffen, mit «subsystem» markierte Packages als Jar zu realisieren.

Die Zusatzinformationen, die eine automatische Transformation von PIM zu PSM zur Durchführung genau solcher Entscheidungen benötigen würde sind hier nicht aufgeführt. Da dieses Kapitel lediglich einige Möglichkeiten für MDA-relevante Modellarten aufzeigen soll, wird auf diesen Aspekt an dieser Stelle auch nicht weiter eingegangen, sondern auf das folgende Kapitel (Transformation) verwiesen.

Schauen wir uns weiterhin an, was aus der Zugriffsbeschränkung auf die Kontodaten geworden ist (Abb. 4.18).

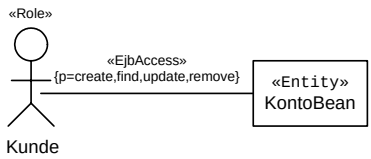


Abb. 4.18
EJB Security

Aus dem Akteur Kunde des CIMs und PIMs ist nun die *Rolle (engl. role)* gleichen Namens geworden, deren Besitzer jetzt den Zugriff auf die Entity Bean KontoBean erhält. Im Sinne der Verwendung einer für die Domäne der Java-EE-Welt spezifischen Terminologie hat sich die Benennung einiger Zugriffsrechte angepasst: während create und update auch in der EJB-Welt die gleiche Bedeutung haben wie im PIM Profil, wird read zu find und delete zu remove. Diejenigen, die bereits einmal mit EJBs gearbeitet haben, werden sich in dieser Umbenennung wiederfinden. An dieser Stelle soll aber auf die „tiefere" Begründung dieses Mappings verzichtet und nur auf das unterliegende Konzept hingewiesen werden: die konsequente Benennung der Modellelemente mit Begriffen aus den jeweiligen Domänen und der Beachtung der eventuellen Möglichkeit, dass diese bei Übergängen zwischen Domänen bzw. Abstraktionsstufen, eventuell angepasst (abgebildet) werden müssen.

Textuelle Artefakte

Schon im vorigen Abschnitt haben wir auf die besondere Klasse der textuellen PSMs wie Quellcode und deklarativen Artefakte wie Deskriptoren usw. hingewiesen. Weil die Erzeugung von Quellcode von Entity Beans und der zugehörigen Interfaces bereits in Kapitel 2

(Überblick und Orientierung) als Beispiel für eine Transformation gezeigt wurde, wollen wir an dieser Stelle auf eine wiederholte Darstellung verzichten und uns einem weiteren Artefakt widmen, das auch textueller Art ist und in der Realisierung der bisher modellierten Aspekte eine wesentliche Rolle spielt: dem Deployment-Deskriptor.

Entscheidet man sich dazu, die modellierte Zugriffsbeschränkung auf `KontoBean` nicht programmatisch innerhalb des Bean-Quellcodes, sondern ausschließlich deklarativ über den Deskriptor zu realisieren, hat dies bei der Erzeugung des Deskriptors folgende Auswirkungen:

- Die Rolle `Kunde` wird definiert
- und die Art der Zugriffsbeschränkung wird festgehalten.

Abb. 4.19 und Abb. 4.20 zeigen die relevanten Ausschnitte aus dem Deskriptor von `KontoBean`. Auch hierbei werden nur die EJB-Prinzipien erklärt, die zum Nachvollziehen des Beispiels unbedingt und minimal notwendig sind.

Abb. 4.19 Auszug Deploymnent-Deskriptor

```
[…]
<session>
  <ejb-name>KontoBean</ejb-name>                    (1)
  <home>KontoHome</home>
    […]
  <security-role-ref>
    <description>
       […]
    </description>
    <role-name>Kunde</role-name>                    (2)
    <role-link>customer</role-link>                 (3)
  </security-role-ref>
  […]
</session>
[…]
```

Betrachtet wir die EJB `KontoBean (1)` mit zugehörigem Home Interface `KontoHome`, das unter anderem die Erzeugung neuer Instanzen von KontoBean kontrolliert. Die Methode `create()` innerhalb des Interfaces realisiert technisch die `create` Funktionalität, für die wir an dieser Stelle exemplarisch die Zugriffsbeschränkung umsetzen wollen, die in Abb. 4.18 festgehalten wurde.

In (2) wird die fachliche Rolle Kunde auf die technische Rolle customer abgebildet. Dadurch wird die Möglichkeit geschaffen auf fachlicher Ebene weiterhin von Kunden (2), auf programmatischer bzw. deklarativer Seite jedoch ab sofort von customer (3) zu reden, da die Verwendung von deutschen Begriffen hier wenig vorteilhaft ist.

```
[…]
<asssembly-descriptor>
[…]
  <security-role>
    <description> […] </description>
    <role-name>customer</role-name>                    (4)
  <security-role>
[…]
<method-permission>
  <role-name>customer</role-name>                      (5)
  <method>
    <ejb-name>KontoBean</ejb-name>                     (6)
    <method-intf>Home</method-intf>                    (7)
    <method-name>create</method-name>                  (8)
  </method>
</method-permission>
[…]
</asssembly-descriptor>
[…]
```

Abb. 4.20 Auszug Deployment- Deskriptor II

In Abb. 4.20 schließlich werden die Zugriffsrechte auf Methodenebene vergeben und mit der entsprechenden Rolle verknüpft: zuerst die Definition der technischen Rolle customer in (4), danach die Verknüpfung der Rolle mit den Rechten auf Methodenebene. Die Methode create (8) des Home Interfaces (7) von KontoBean (6) wird für die Rolle customer (5) freigeschaltet.

Da die create Methode die Erzeugung von KontoBeans regelt ist damit der geforderte Sachverhalt aus Abb. 4.18 für die create-Permission erledigt. Die Realisierung der Funktionalitäten find, update und remove kann prinzipiell auf ähnliche Weise geschehen. Da dies aber kein Buch über EJBs, sonder über die Model-Driven Architecture ist, werden wir diese Umsetzungen nicht weiter verfolgen.

Nur eins sei noch erwähnt: Obwohl in unserem Beispiel nicht verwendet, haben wir mit dem eben vorgestellten Konzept gleichzeitig eine mögliche Umsetzung der modellierten Zugriffsbeschränkung auf Operationen (~Methoden) für die Plattform JavaEE realisiert, die in Abb. 4.7 durch den Stereotypen «ExecutionPermission» gefordert wurde. Zur Realisierung müssten für die betroffenen Elemente in Abb. 4.20 lediglich der Name der Bean, die die entsprechende Operation enthält, eingetragen (5), in (7) das Interface, in dem die Operation definiert wurde, angegeben, und in (8) schließlich der Name der markierten Operation übernommen werden.

Literatur

[CN01] Paul Clemens und Linda Northrop: *Software Product Lines – Practices and Patterns*. Addison-Wesley, 2001.

[EJB] Sun Microsystems: *Enterprise JavaBeans Technology*.
http://java.sun.com/products/ejb/ (letzter Abruf Mai 2005)

[Eva03] Eric Evans: *Domain-Driven Design – Tackling Complexity in the Heart of Software*. Addison-Wesley, 2003.

[Fra03] David S. Frankel: *Model Driven Architecture – Applying MDA To Enterprise Computing*. John Wiley & Sons, 2003.

[GOF] Erich Gamma, Richard Helm, Ralph E. Johnson und John Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gom04] Hassan Gomaa: *Designing Software Product Lines with UML – From Use Cases to Pattern-Based Software Architectures*. Addison-Wesley, 2004.

[KWB03] Anneke Kleppe, Jos Warmer und Wim Bast: *MDA Explained. The Model Driven Architecture – Practice and Promise*. Addison-Wesley, 2003.

[MDAGuide] Object Management Group (OMG): *MDA Guide Version 1.0.1*.
http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf (letzter Abruf Mai 2005)

[Oes05] Bernd Oestereich: *Objektorientierte Geschäftsprozessmodelllierung und modellgetriebene Softwareentwicklung*. In: HMD – Praxis der Wirtschafts-

informatik, Heft 241, S. 27-34, dpunkt, Februar 2005.
http://www.oose.de/downloads/HMD_241_GPM_und_MDSE_Oestereich.pdf (letzter Abruf Mai 2005)

[OWS+03] Bernd Oesterreich, Christian Weiss, Claudia Schröder, Tim Weilkiens und Alexander Lenhard: *Objektorientierte Geschäftsprozessmodellierung mit der UML*. dpunkt, 2003.

[VM] Wikipedia: *Virtual Machine*.
http://en.wikipedia.org/wiki/Virtual_Machine (letzter Abruf Mai 2005)

5 Transformation

„Ullus res muta. Anima irreti. PIM ad PSM transmuta. PSM ad Code.“

Überliefert aus den Anfängen der Modelltransformation

In bisherigen, codezentrierten Ansätzen der Softwareentwicklung sind Modelle – falls überhaupt – vornehmlich zu Veranschaulichungs- und Dokumentationszwecken verwendet worden. Die MDA möchte hier mehr: Insbesondere sollen Modelle ineinander überführt werden können und direkt der Anwendungsentwicklung dienen. Wesentlich zur Durchführung des MDA-Ansatzes ist daher die Transformation von Modellen, ein Konzept, dem sich das vorliegende Kapitel widmet.

5.1 Einführung

In Kapitel 4 haben wir die verschiedenen Modellklassen der MDA kennen gelernt. Ebenfalls dort dargestellt wurde, dass diese Modellklassen auf verschiedenen Abstraktionsebenen angesiedelt sind, dabei jedoch in genau definierter Beziehung zueinander stehen.

So beschreiben PIM und PSM das zu entwickelnde Softwaresystem von verschiedenen Standpunkten aus: den plattform-unabhängigen und den plattform-spezifischen. Wie wir weiter gesehen haben, hat dies den Zweck, von den technologiespezifischen Aspekten eines Systems zu abstrahieren. Oder in umgekehrter Richtung: das plattformunabhängige Modell soll nach und nach um genau diese Aspekte verfeinert werden, bis eine lauffähige Anwendung entsteht. Diesen Verfeinerungsvorgang bezeichnet man als *Transformation* eines PIM in ein PSM.

Modell-zu-Modell Transformation (M2M)

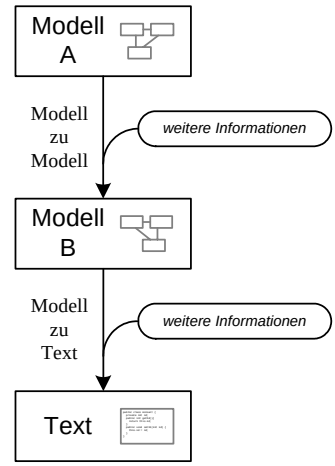
Abstrahieren wir zusätzlich von der spezifischen Information, die PIM und PSM tragen, können wir allgemein von einer *Modell-zu-Modell Transformation (M2M)* sprechen. Wir werden im Anschluss an dieses Unterkapitel sehen, dass es neben der PIM zu PSM Transformation Anwendungsfälle für Modelltransformationen gibt, die diese Abstraktion rechtfertigen.

Bei der Überführung eines PSM in Quellcode verhält es sich ähnlich wie bei der von PIM in PSM. In analoger Weise *kann* eine Verfeinerung mit ihr vollzogen werden[1]. Rufen wir uns zunächst in Erinnerung, dass man Quellcode auch als Modell betrachten kann (vgl. Abschnitt 3.1.2), so liegt der Schluss nahe, auch bei dieser Überführung von einer Modell-zu-Modell Transformation zu sprechen.

Modell-zu-Text Transformation (M2T)

Allerdings ist mit einer solchen Art der Überführung eine Umformung in eine textuelle Repräsentation verbunden, für die sich zusätzliche Anforderungen ergeben, sodass wir sie unter dem Titel *Modell-zu-Text Transformation (M2T)* gesondert erörtern werden und ihr ein eigenes Unterkapitel zueignen.

Abb. 5.1 Transformationsvarianten



Beiden Transformationsvarianten – Modell-zu-Modell und Modell-zu-Text – ist gemein, dass man sie sowohl manuell, teil- oder vollautomatisiert umsetzen kann. Wir werden uns insbesondere der

[1] Wir werden später zeigen, dass es in der Regel sinnvoll ist ein „Quellcode-Modell“ zu erstellen, welches gleichzeitig eine abstrakte Sicht auf den Quellcode liefern und die Rolle eines PSM einnehmen kann.

letzteren Ausprägung zuwenden und uns nachfolgend der diesbezüglichen Fragestellung widmen, wie die in Abb. 5.1 angedeuteten *weiteren Informationen* dazu festgehalten werden müssen.

5.2 Anwendungsfälle für Transformationen

Bevor wir uns näher mit der Technik hinter der Transformation befassen, wollen wir zunächst auf das Versprechen von oben zurückkommen und uns eine Übersicht darüber verschaffen, unterschiedliche Zwecke Transformationen erfüllen können:

Verfeinerungen

Bei dem oben bereits angesprochenen zentralen Transformationsmuster von PIM zu PSM wird das PIM mit plattformspezifischen Informationen angereichert, also verfeinert. *Verfeinerungen* sind darüber hinaus beispielsweise bei einer Überführung von PIM zu PIM möglich, etwa beim Übergang von einem Analyse- zu einem Design-Modell (siehe Kapitel 4).

Abstraktion

Die Umkehrung der Verfeinerung ist die *Abstraktion*. Eine entsprechende Transformation kommt dann zur Anwendung, wenn *auf das Wesentliche* fokussiert werden soll, also z. B. auf die sicherheitsrelevanten Details eines Softwaresystems. Auch im Zuge einer Legacy-Integration bzw. eines Reengineerings kann ein solcher Transformationsschritt sinnvoll sein. Dabei möchte man beispielsweise Schnittstellen aus vorhandenem Quellcode oder sogar binären Executables gewinnen, und diese aus den erstellten Modellen bzw. der resultierenden Anwendung heraus ansprechen.

Migration

Bereits kleinste Änderungen an einer technologischen Plattform, beispielsweise im Zuge eines Versionswechsels, kann eine *Migration* eines gesamten Softwaresystems bedingen. Alle Bestandteile des Systems, also auch ein etwaiger Datenbestand, müssen dabei so transformiert werden, dass sie weiterhin ohne jegliche Einschränkung korrekt funktionieren. Eine mitunter sehr komplexe Form der Migration ist die Technologiemigration, etwa bei einem Wechsel von COBOL nach Java.

Refaktorisierung

Eine Änderung an der internen Struktur einer Software, die der Erhöhung ihrer Verständlichkeit und der Verbesserung ihrer Architektur dient, ihre Funktionalität – genauer gesagt ihr nach außen sichtbares Verhalten – aber unberührt lässt, nennt man *Refaktorisierung (engl. refactoring)* [Fow99]. Teil einer Refaktorisierung kann

beispielsweise die Herausfaktorisierung von Gemeinsamkeiten (etwa durch Umstrukturierung oder Generalisierung) sein.

Optimierung

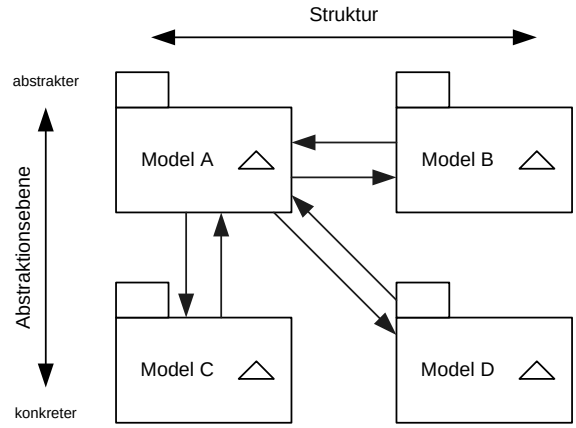
Durch eine *Optimierung* soll der Platz- und/oder Rechenzeitbedarf eines Programms verringert werden. Wie bei der Refaktorisierung sind Quelle und Ziel bezüglich ihrer Funktionalität äquivalent. Die Abgrenzung zur Refaktorisierung ist überdies nicht immer ganz trennscharf, denn selbige kann eine ähnliche Zielsetzung verfolgen. Während eine Refaktorisierung aber in erster Line aufgrund von Erfahrungswerten eines Entwicklers von Hand angestoßen wird, wird die Optimierung automatisiert und bezogen auf eine Zielplattform ausgeführt.

Darstellungsumformung

Eine *Darstellungsumformung* ändert weder die interne Struktur noch das externe Verhalten einer Software, sie betrifft allein ihre Repräsentation. Beispielhaft sei hier die Umformung von einer textuellen in eine graphische Notation angeführt.

Eine Transformation lässt sich nicht immer eindeutig einer der genannten beschriebenen Ausprägungen zuordnen. Beispiel: Eine Transformation, die zugleich verfeinernd als auch optimierend wirkt.

Abb. 5.2 Klassifikation von Transformationen



Eine weitere übliche Klassifizierungsmöglichkeit für Transformationen wird an der Struktur und der Abstraktionsebene von Modellen festgemacht. Man unterscheidet vertikale, horizontale und schiefe Transformationen. *Vertikale Transformationen* haben dabei eine Änderung der Abstraktionsebene zur Folge, lassen dabei aber die Struktur intakt, während *horizontale Transformationen* auf der gleichen Abstraktionsebene stattfinden, aber die Struktur verändern.

Schiefe Transformationen (engl. oblique transformations) ändern sowohl Struktur also auch Abstraktionsebene. Abb. 5.2 hält diese Klassifikation anschaulich fest.

Die Ausprägungen *Verfeinerung* und *Abstraktion* sind demnach vertikale und *Migration*, *Refaktorisierung* und *Optimierung* horizontale Transformationen. Die Variante *Darstellungsumformung* lässt sich hingegen nicht in dieses Schema pressen.

5.3 Modell-zu-Modell Transformationen

Betrachten wir nun die generische Transformation eines Modells in ein anderes. In Kapitel 3 haben wir erörtert, dass ein Modell in einer Modellierungssprache verfasst, und die Syntax dieser Sprache im zugehörigen Metamodell festgehalten wird. Dort haben wir auch gesehen, dass man ein Modell als Instanz eines Metamodells betrachten kann.

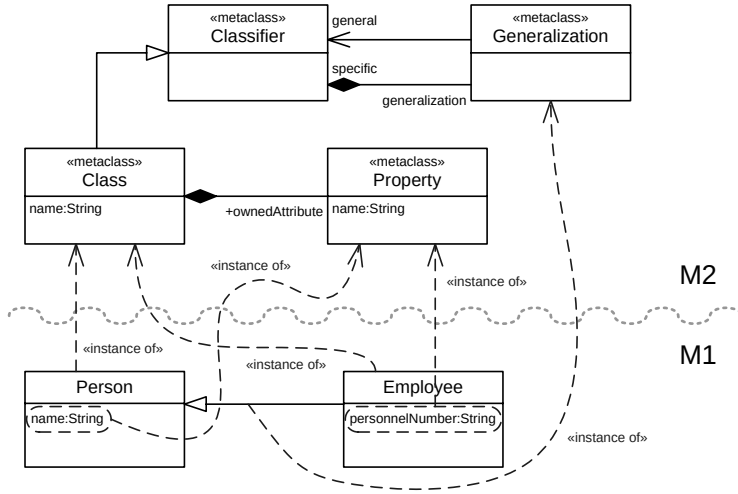


Abb. 5.3 Ein Modell als Instanz eines Metamodells

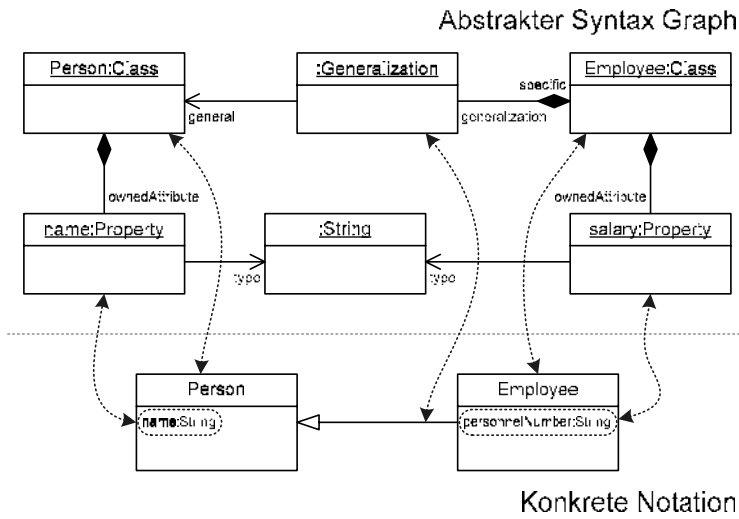
Abb. 5.3 veranschaulicht noch einmal diesen Zusammenhang. Die Abbildung zeigt hierzu einen Ausschnitt eines auf Metalevel M1 (Modell) angesiedelten UML Klassendiagramms und den dazu korrespondierenden Teil des UML 2.0 Metamodells [UML2SS] auf Metalevel M2 (Metamodell). Die gestrichelten `«instance of»`-Pfeile verbinden dabei die M1-Instanzelemente mit den M2-Metaklassen. So sind etwa die Attribute einer Klasse Instanzen der Metaklasse `Property`, die Generalisierungsbeziehung Instanz einer Metaklasse `Generalization`, und so fort.

Des Weiteren finden wir den Namen einer Klasse als Attribut `name:String` an der Metaklasse `Class`. Die Assoziationen zwischen der Metaklasse `Classifier` sowie der Metaklasse `Generalization` spezifizieren die jeweilige Rolle, die eine Klasse in einer Generalisierungsbeziehung spielen kann (`specific` oder `general`, im Beispiel `Employee` bzw. `Person`).

Abstrakter Syntax Graph als IMR eines Modells

Alternativ zu dieser Darstellungsvariante kann man die Instanz-Beziehung auch in einem Objektdiagramm visualisieren, so wie in Abb. 5.4 geschehen. Hier wird besonders deutlich, dass die *In Memory Representation (IMR)* eines Modells die Form eines Objektgraphen besitzt – in diesem Zusammenhang spricht man daher auch vom *abstrakten Syntax Graph (ASG)*.

Abb. 5.4 Ein Modell als Instanz eines Metamodells



Die beiden Abbildungen verdeutlichen noch einmal, dass sich das Instanzierungs- bzw. Typisierungs-Konzept, welches die Metalevel M0 (System) und M1 (Modell) verbindet, auch auf die Metalevel M1 und M2 übertragen lässt. Ein Element eines Modells ist also nichts anderes als das instanziierte Objekt einer Klasse eines Metamodells.

5.3.1 Das Schema metamodellbasierter Modelltransformationen

Metamodelle liefern uns den Schlüssel, um Modelle ineinander transformieren zu können, indem wir sie als Basis für eine Abbil-

dungsfunktion zwischen ihnen heranziehen. Abb. 5.5 skizziert diese Idee und zeigt das prinzipielle Schema einer solchen metamodellbasierten Modelltransformation.

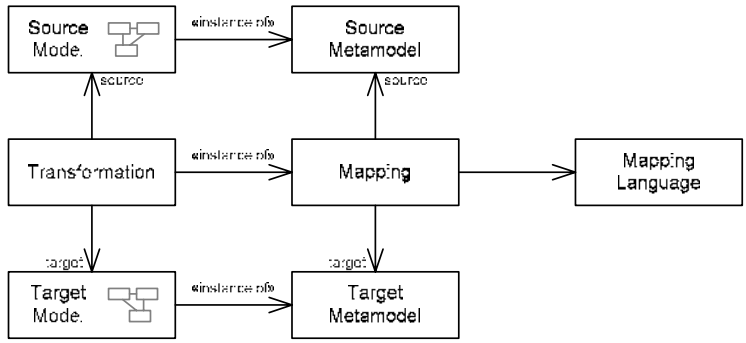


Abb. 5.5 Schema einer metamodellbasierten Modelltransformation

Transformationsregeln – hier als Mapping bezeichnet – halten die Transformation formal fest. Sie beschreiben, wie Elemente aus dem Quellmodell auf Elemente des Zielmodells abgebildet werden, und stützen sich dazu auf das Vokabular des Quell- bzw. Zielmetamodells. Die Regeln selbst sind in einer Sprache verfasst, auf die wir uns vorerst nicht festlegen wollen.

Nehmen wir zunächst einmal an wir wollten unsere Transformationsregeln in einer gängigen Programmiersprache verfassen. In Kapitel 3 hatten wir bereits konzis angerissen, wie man an ein Programmiermodell für den Zugriff auf Modelle bzw. Metamodelle gelangen kann. Kurzerhand zur Erinnerung: das Zauberwort dazu lautet MOF (Meta Object Facility). Ihre Spezifikation [MOF2] bzw. ihre Adjazenten wie etwa MOF2IDL [MOF2IDL] und JMI [JMI] beschreiben wie ihre Konzepte auf programmiersprachliche Konzepte (CORBA respektive Java) abgebildet und damit implementiert werden können.

Reflektives Programmiermodell

Da das aus einer solchen Abbildung resultierende Programmiermodell die Metainformationen eines Modells *widerzuspiegeln* vermag, nennt man es auch *reflektives Programmiermodell*.

Für den Rest dieses Kapitels wollen wir diese Funktionalität als gegeben betrachten und bezüglich weiterer Details auf die praktische Vertiefung im zweiten Teil dieses Buches verweisen. Auf die ebenfalls verwandten Spezifikation [MOF2XMI], die wir gleichfalls in Kapitel 3 beschrieben hatten, gehen wir hingegen – in groben Zügen – noch einmal ein, da sie uns bei Eingrenzung und Verständnis unseres Diskursbereichs helfen.

Die XMI-Spezifikation beschreibt, wie man Modelle respektive Metamodelle auf XML-Strukturen abbildet und betrachtet auch die umgekehrte Richtung. Genauer: XMI spezifiziert, wie MOF-basierte Metamodelle – also Instanzen des MOF-Metamodells – als Dokumenttypdefinitionen (DTD) bzw. ab XMI 2.0 als XML Schema Definitionen (XSD) festgehalten werden können. Eine Instanz eines solchen Metamodells (also ein Modell) kann damit als XML-Dokument formuliert werden, das dieser Definitionen entspricht.

Da die MOF mit sich selbst definiert ist, kann auch die MOF selbst die Position des abzubildenden Metamodells einnehmen, also ein MOF-basiertes Metamodell in die Rolle des Modells schlüpfen und selbst als XML-Dokument abgebildet werden.

Kurzum erhält man damit einen Mechanismus um (Meta-)Modelle, die als Objekte im Speicher vorliegen, in einem standardisierten Austauschformat persistierbar zu machen. Schauen wir uns dazu Abb. 5.6 an. Die Abbildung veranschaulicht die folgenden drei Schritte, wie sie üblicherweise als Beiwerk bei einer Modell-zu-Modell Transformation durchgeführt werden müssen:

(1) Zunächst wird das Modell deserialisiert, d. h. es wird von seiner persistenten Form (hier im XMI-Format) in eine *In Memory Representation (IMR)*[2] gebracht.

(2) Danach wird auf der IMR eine Transformation ausgeführt. In der Abbildung wird beispielhaft das öffentliche Attribut `attr1` in ein Privates mit gleichem Namen samt zugehörigen getter/setter-Paar überführt.

(3) Das Modell wird wieder serialisiert, d. h. von seiner IMR in einen persistenten Zustand gebracht (wiederum im XMI-Format).

Zunächst einmal ist anzumerken, dass die Wahl von XMI als Dateiformat, so wie in der Abbildung wiedergegeben, keineswegs in irgendeiner Form bindend ist. XMI gehört aber zu den OMG-Standards im Dunstkreis der MDA, ist weit verbreitet und damit erst einmal ein naheliegender Verdächtiger. Zudem bringen die weiter oben erwähnten MOF-Implementierungen oftmals bereits die Möglichkeit mit, XMI zu lesen und zu schreiben.

[2] Zur Verdeutlichung, dass sich die in Abb. 5.6 dargestellten Klassen in einer IMR befinden, sind diese gestrichelt dargestellt.

Ergo können wir mithilfe einer solchen Implementierung von der Deserialisierung aus Schritt 1 sowie der Serialisierung aus Schritt 3, abstrahieren. (Die De-/Serialisierung entspricht im Übrigen der unter 5.2 beschriebenen *Darstellungsumformung*.)

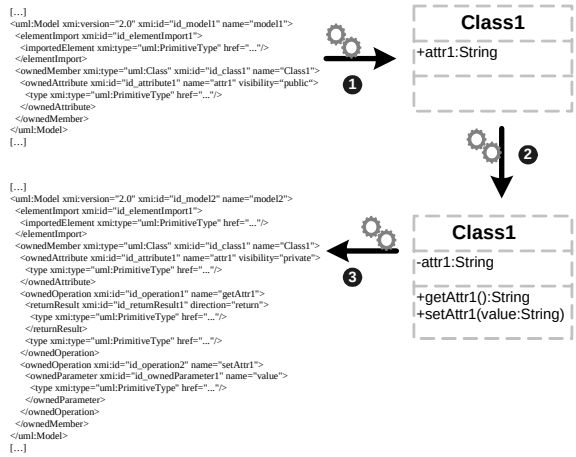


Abb. 5.6 Modell De-/Serialisierung

Das werden wir für den Rest dieses Unterkapitels auch tun und uns stattdessen mit Schritt 2 näher befassen, da er insbesondere für die PIM → PSM Transformation von Bedeutung ist.

5.3.2 Beispiel: UML 2.0 PIM zu Java PSM

Abb. 5.7 zeigt eine PIM → PSM → Code Transformation in der Übersicht. In diesem Unterkapitel interessiert uns zunächst der Übergang von PIM zum PSM. Wie in der Abbildung zu sehen ist soll unsere Transformation vorwärts gerichtet ausgelegt sein. Zudem beschränken wir uns auf eine 1:1 Beziehung zwischen PIM und PSM.

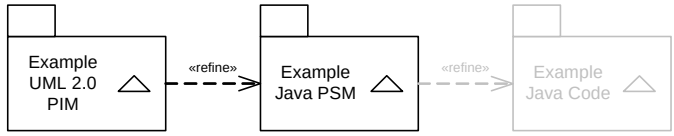
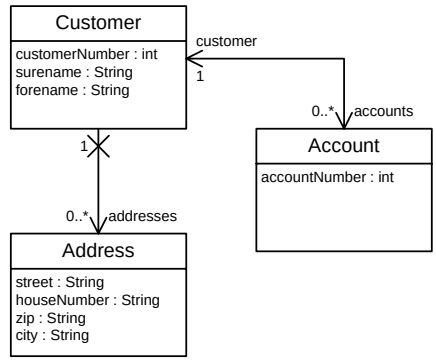


Abb. 5.7 Schritt 1: UML 2.0 PIM zu Java PSM

Als Quellmetamodell wählen wir das UML 2.0 Metamodell, genauer gesagt einen Ausschnitt aus dem Bereich *Classes* [UML2SS]. Ziel unserer Transformation soll ein Programm in der Programmiersprache Java sein. Dementsprechend haben wir in einem ersten Schritt unser UML 2.0 PIM in ein Java PSM zu überführen.

Wir haben Quell- und Zielsprache mit folgendem Hintergedanken gewählt: sie sind zum einen einem breiten Publikum bekannt und liegen zum anderen bezüglich Syntax und Semantik nahe beieinander. Letzteres soll uns dabei helfen, dass die Komplexität der Transformationsregeln uns nicht vom Wesentlichen ablenkt, und wir uns voll auf die Technik der Transformation konzentrieren können.

Abb. 5.8 Beispielmodell



Für die weiteren Betrachtungen ziehen wir das in Abb. 5.8 dargestellte Beispielmodell heran. Natürlichsprachlich formuliert gibt das Modell den folgenden Sachverhalt wieder:

- Es gibt drei Klassen: Kunde, Adresse und Konto.
- Ein Kunde hat die Attribute Kundennummer, Vorname und Nachname.
- Ein Konto hat das Attribut Kontonummer.
- Eine Adresse hat die Attribute Straße, Hausnummer, Postleitzahl und Ort.
- Es gibt eine Assoziation zwischen Kunde und Konto die besagt, dass genau einem Kunden eine beliebige Anzahl von Konten zugeordnet sein kann. Die Assoziation ist sowohl von Kunde als auch von Konto navigierbar.
- Eine weitere Assoziation zwischen Kunde und Adresse spezifiziert, dass genau einem Kunden beliebig viele Adressen zugeordnet sein können. Diese Assoziation ist nur von einem Kunden zur Adresse navigierbar und nicht umgekehrt.

Wenn ein Java-Entwickler die UML lesen kann, dann „weiß“ er, wie er ein solches Modell in Java umsetzen kann. Die Regeln dazu hat er

vielleicht nicht explizit formuliert, er kann aber die Konzepte aus der einen Sprache in die der anderen übertragen. Schauen wir uns *ein* mögliches Endergebnis einer solchen Übersetzung an (Abb. 5.9).

Abb. 5.9 Beispiel Java-Klassen

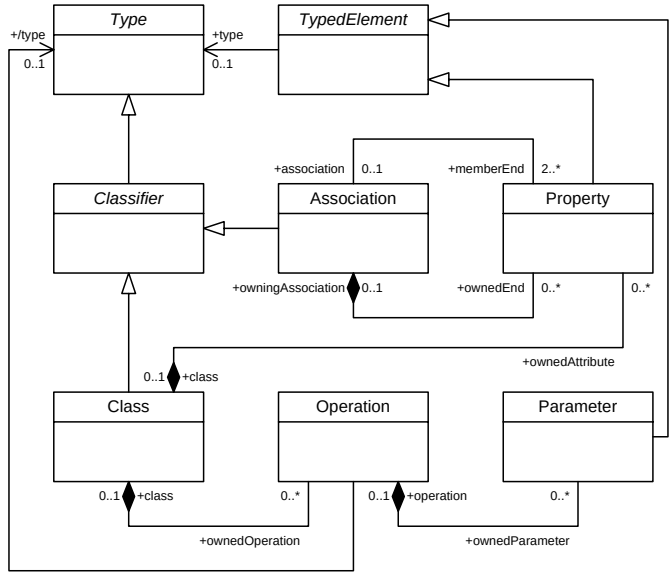
```
public class Customer {
  int customerNumber;
  String surename;
  String forename;
  List accounts;
  List addresses;
}
```

```
public class Account {
  int accountNumber;
  Customer customer;
}
```

```
public class Address {
  String street;
  String houseNumber;
  String zip;
  String city;
}
```

Der Abbildung ist zu entnehmen, dass hier Annahmen getroffen worden sind, die die Umsetzung der Assoziationen zwischen den Klassen betreffen. So wurden die Assoziationsenden, die eine Multiplizität größer 1 aufweisen als Listen (Java Klasse `List`) realisiert. Wir sehen also bereits, dass sich nicht alle Konzepte 1:1 übertragen lassen.

Abb. 5.10 Ausschnitt UML 2.0 Metamodell



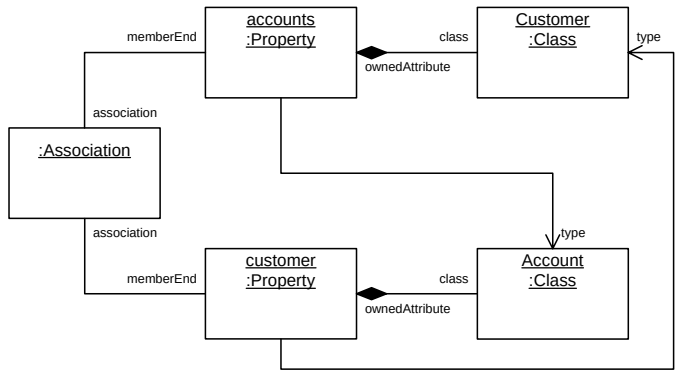
Aber der Reihe nach. Eingangs haben wir beschrieben, dass wir unsere Transformation automatisiert auf Basis von Metamodellen durchführen wollen. Betrachten wir dazu den folgenden Ausschnitt aus unserem Quellmetamodell, dem UML 2.0 Metamodell (siehe Abb. 5.10).

Die Darstellung beschreibt die Konzepte, die wir in unserem Beispielmodell aus Abb. 5.8 verwendet haben in abstrakter Form mit den Mitteln der UML selbst (UML Infrastructure). Sie zeigt dazu die Elemente Typ, Typisiertes Element, Klasse, Classifier, Attribut (bzw. Eigenschaft), Assoziation, Operation und Parameter sowie ihren strukturellen Zusammenhang.

Aber welche Elemente unseres Beispielmodells sind welchen Elementen des Metamodells zugeordnet? Dazu muss man die Spezifikation der OMG studieren und nachschlagen welche Notation die jeweiligen Metamodellelemente haben (siehe [UML2SS]).

Den so ausgemachten Zusammenhang von Modell und Metamodell kann man anschaulich in einem Objektdiagramm darstellen. In Abb. 5.11 haben wir dies für die Klassen `Customer` und `Account` sowie der zwischen diesen Klassen bestehenden Assoziation getan.

Abb. 5.11 Beispiel Objektdiagramm



Wir sehen, dass sich hinter der Notation einer Assoziation aus unserem Beispieldiagramm nicht nur ein Element, sondern derer drei befinden. Neben der Assoziation selbst sind dies die Attribute bzw. Eigenschaften, die die Assoziation mit den jeweiligen Klassen verbindet.

Damit haben wir die Bausteine für unser Quellmodell zusammen. Als Nächstes widmen wir uns unserer Zielplattform. Dazu schauen

wir uns das Endergebnis der Klasse `Customer` aus Abb. 5.9 einmal im Detail an. Abb. 5.12 zeigt einen Ausschnitt aus dem entsprechenden Java-Quellcode.

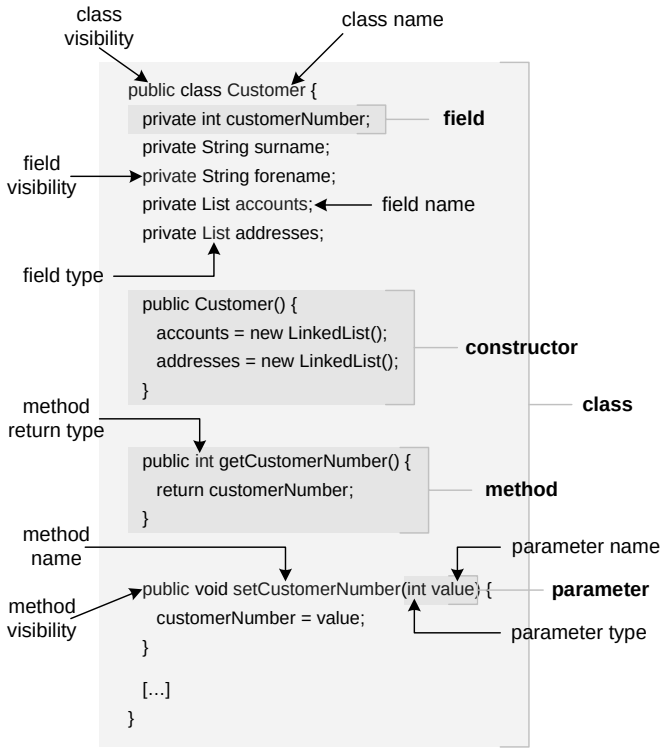


Abb. 5.12 Beispiel Java Klasse im Detail

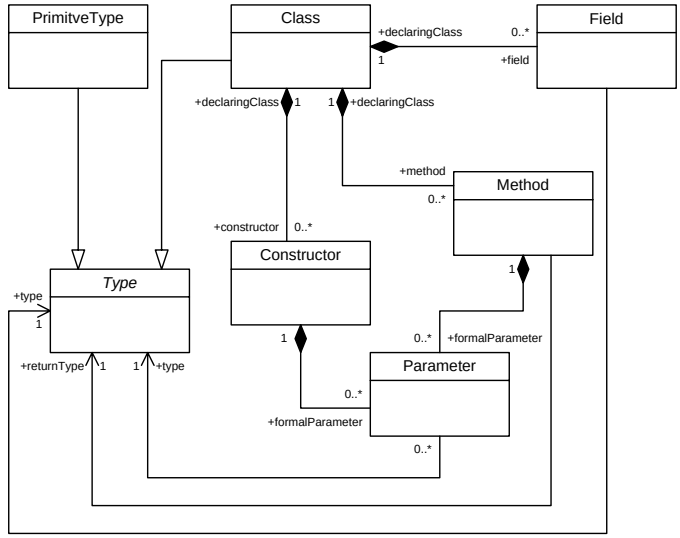
Wir haben für Sie die Klasse in ihre Bestandteile zerlegt, diese benannt und in Beziehung gesetzt. Dabei wurde von den einzelnen Elementen auf allgemeine Konzepte geschlossen. Die resultierenden Konzepte sind in einem Zielmetamodell festgehalten. Das Resultat, zu sehen in Abb. 5.13, birgt indes keine Überraschungen. Bereits eingangs hatten wir erwähnt, dass UML und Java beide objektorientiert sind und damit nah beieinander liegen.

Das beschriebene induktive Vorgehen wurde allein der Anschaulichkeit wegen gewählt. In der Praxis ist durch die Komplexität von Zielplattform bzw. -sprache ein solches Vorgehen in der Regel zu aufwendig und damit nicht praktikabel.

Aber wie kommt man dann an ein entsprechendes Metamodell? Entweder man hat das Glück, dass sich schon jemand die Mühe gemacht hat und ein adäquates Metamodell aufgestellt hat, oder man

beschafft sich die passende Plattform-/Sprachspezifikation und setzt das Metamodell anhand dieser eigenhändig auf (für unser Beispiel wäre die *Java Language Specification (JLS)* [JLS] heranzuziehen gewesen).

Abb. 5.13
Java Metamodell



Quell- und Zielmetamodell haben wir damit vorliegen. Begeben wir uns nun an die Formulierung der Transformationsregeln:

Zunächst wollen wir uns dieser Problemstellung natürlichsprachlich nähern. Die Regeln lauten demnach:

1. Zu jeder Klasse im Quellmetamodell (QMM) erstelle eine Klasse im Zielmetamodell (ZMM). Übernehme den Namen der Klasse sowie ihre Sichtbarkeit.
2. Zu jedem Attribut einer Klasse aus dem QMM erstelle ein Feld im ZMM. Übernehme den Namen des Attributs, den Typ sowie dessen Sichtbarkeit.
3. Zu jeder Operation einer Klasse im QMM erstelle eine Methode im ZMM. Übernehme den Namen der Operation, deren Rückgabetyp sowie ihre Sichtbarkeit.
4. Zu jedem Parameter einer Operation im QMM erzeuge einen Parameter im ZMM. Übernehme den Namen des Parameters und dessen Typ.

Abb. 5.14 zeigt diese Regeln in einer schematisch grafischen Darstellung. Die Rechtecke links und rechts repräsentieren die Elemente des Quell- bzw. des Zielmetamodells, das Hexagon in der Mitte enthält den Namen der Transformationsregel. Obwohl in unserem Falle eine 1-zu-1 Transformation vorliegt (für jedes Quell-Element wird genau ein Zielelement erzeugt), ist dies keine generelle Bedingung an eine Transformation.

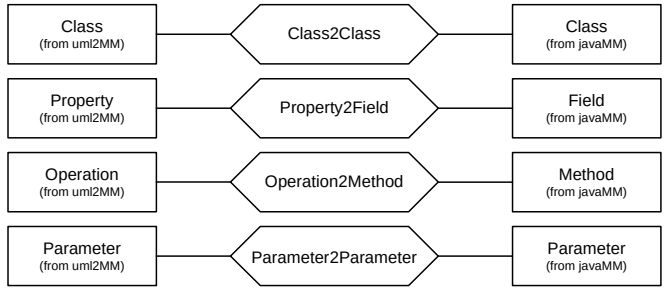


Abb. 5.14 Übersicht Transformationsregeln

Abstrahiert man diesen am Beispiel erläuterten Sachverhalt, bestehen Transformationsregeln aus den folgenden zwei Bestandteilen:

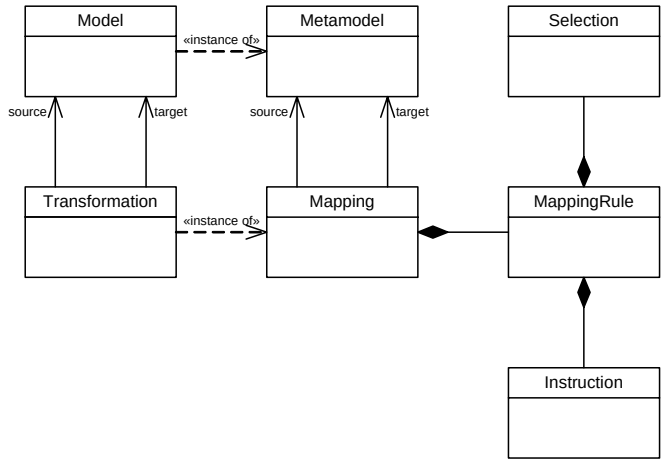
- einer Selektion von Elementen aus dem Quellmodell
- und einer Menge von Instruktionen (in unserem Beispiel waren dies die Anweisungen „erzeuge“ und „setze Wert“) Grundsätzlich sind aber auch Löschoperationen oder beliebige andere Operationen denkbar.

Stellt man die gefundenen Ergebnisse in einem schematischen Modell dar – man erhält dann quasi ein Transformationsmetamodell – könnte dies wie in Abb. 5.15 dargestellt aussehen:

Alle Modelle sind Instanzen ihrer zugrunde liegenden *Metamodelle*. Soll ein Modell aus der Transformation eines anderen entstehen, wird eine Beschreibung benötigt, wie die Elemente des neuen Modells zu Mengen der Elemente des Quellmodells in Beziehung stehen. Die Menge dieser Beziehungen wollen wir als *Mapping* zwischen beiden Metamodellen verstehen. Eine *Regel* eines solchen Mappings besteht aus der Angabe der *Bedingung*, die eine Menge von Elementen des Quellmodells erfüllen muss, um an der Transformation teilzunehmen (der *Selektion*) sowie der Menge von *Anweisungen*, die auf den so ermittelten Elementen auszuführen sind (also der Menge der *Instruktionen*). Eine *Transformation* letztend-

lich besteht aus der Angabe einer Teilmenge der so spezifizierten Regeln, und stellt eine Instanz des Mappings dar.

Abb. 5.15 Transformationsmetamodell



An dieser Stelle steht uns schon ein ordentlicher theoretischer Unterbau zum Themenkomplex Transformation zur Verfügung. Wie man aufbauend auf dieser Basis zu konkreten Umsetzung kommt, soll nun weiter verfolgt werden.

5.3.3 Implementierungs-Strategien für Transformationen

In der Praxis existiert eine Vielzahl von Ansätzen, die die in den vorigen Abschnitten beschriebenen Transformationen implementieren. Im Folgenden wollen wir diejenigen herausgreifen, die in der Praxis am häufigsten anzutreffen sind.

Einsatz universeller Programmiersprachen

Die Möglichkeit, die intuitiv wohlmöglich zuerst in Betracht gezogen wird, wenn man sich Gedanken um die Realisierung einer Transformation macht, fällt in die Klasse der Umsetzung mittels einer gängigen, universellen Programmiersprache. Einzelschritte einer Transformation werden in Befehlsfolgen übersetzt und einzelne Regeln durch die Bündelung in Methodenaufrufe gekapselt. Abb. 5.16 zeigt ein Beispiel für eine solche Umsetzung in der Programmiersprache Java.

Die so entstandenen „Regel-Methoden“, in der gewünschten Reihenfolge und mit den richtigen Parametern aufgerufen, ergeben dann die vollständige Ausführung der Transformation.

```
public javaMM.Class class2class(uml2MM.Class uml2Class) {
  […]
  javaMM.Class javaClass = new javaMM.Class();
  javaClass.setName(uml2Class.getName());
  javaClass.setVisibility(uml2Class.getVisibility());
  List list = uml2Class.getOwnedAttributes();
  for (Iterator it = list.iterator(); it.hasNext();) {
    uml2MM.Property uml2Property = (uml2MM.Property)it.next();
    javaMM.Field javaField = property2field(uml2Property);
    javaClass.getFields().add(javaField);
  }
  […]
  return javaClass;
}

public javaMM.Field property2field(uml2MM.Property uml2Property) {
  […]
  javaMM.Field javaField = new javaMM.Field();
  javaField.setName(uml2Property.getName());
  javaField.setVisibility(uml2Property.getVisibility());
  javaField.setType(uml2Property.getType());
  […]
  return javaField;
}
```

Abb. 5.16 Umsetzung einer Transformation in Java

Dieser Ansatz hat jedoch zwei wesentliche Nachteile:

- Änderungen können nur direkt im Quellcode der gewählten Programmiersprache vorgenommen werden und werden so eventuell erst nach der erneuten Übersetzung des transformierenden Codes wirksam.
- Zur Beschreibung und Übersetzung der Transformation sind gute Kenntnisse der eingesetzten Programmiersprache erforderlich, wobei oft auch Features dieser Sprache verstanden werden müssen, die mehr in den Anforderungen der Sprache selbst, als in denen der Transformationsbeschreibung liegen. Dieser Sachverhalt verletzt somit das Ziel der MDA möglichst domänenspezifische Sprachen (DSLs) einzusetzen, die speziell auf ihre Aufgaben zugeschnitten sind.

Einsatz von Skriptsprachen

Eine Möglichkeit, die Neuübersetzung des transformierenden Programms nach Änderungen in der Transformationsbeschreibung zu umgehen, und auch die Bequemlichkeit der Anwendung durch Beschränkung der Sprachkonstrukte und Aufweichen sprachresultierender Restriktionen (z. B. strikte Typisierung) für den Anwender zu

erhöhen, ist die Wahl einer Skriptsprache zur Transformations-Durchführung.

Wesentliche Merkmale von Skriptsprachen sind, dass sie ohne expliziten Compilelauf auskommen, und oftmals Features besitzen, die den einfacheren Umgang mit Objekten und Datenstrukturen erlauben. Ein Beispiel für eine solche Skriptsprache ist *Groovy* [Groovy] – zu den Vorteilen von Groovy zählen:

- der interpretierte Charakter der Sprache, wodurch kein expliziter Compilelauf nach Änderungen notwendig wird,
- die native Syntax zur Behandlung von komplexen Datenstrukturen – bspw. Collections und Maps – wie etwa zur automatischen Iteration (`attributes.each`, etc.),
- die native Unterstützung Regulärer Ausdrücke
- Introspektion, etc.

Abb. 5.17 Transformation mit Groovy

```
javaMM.Class class2class(uml2MM.Class uml2Class) {
  […]
  javaClass = new javaMM.Class()
  javaClass.name = uml2Class.name
  javaClass.visibility = uml2Class.visibility
  uml2Class.ownedAttributes.each { uml2Property |
    javaField = property2field(uml2Property)
    javaClass.fields << javaField
  }
  […]
  return javaClass
}

javaMM.Field property2field(uml2MM.Property uml2Property) {
  […]
  javaField = new javaMM.Field()
  javaField.name = uml2Property.name
  javaField.visibility = uml2Property.visibility
  javaField.type = uml2Property.type
  […]
  return javaField
}
```

Abb. 5.17 zeigt ein Beispiel für eine Groovy-Transformation. Ohne weiter auf Details einzugehen, wird intuitiv deutlich, dass die Lesbarkeit der Beschreibung vereinfacht wird.

Einsatz einer Transformationssprache

Die in diesem Buch favorisierte Umsetzung einer Transformationsbeschreibung liegt in der Verwendung einer dedizierten *Transformationssprache*. Diese erfüllt nicht nur die Forderung nach Änderbarkeit ohne Neuübersetzung des transformierenden Programms, sondern zusätzlich die Anforderungen an eine DSL: sie erfordert nur

die Kenntnisse, die zur Beschreibung einer Transformation unbedingt notwendig sind.

```
module uml2MM2javaMM;
create OUT:javaMM from IN:uml2MM;

[…]

rule Class2Class {
  from uml2Class : uml2MM!Class
  to javaClass : javaMM!Class
  mapsTo uml2Class (
    name <- uml2Class.name,
    visibility <- uml2Class.visibility
  )
}

rule Property2Field {
  from uml2Property : uml2MM!Property
  to javaField : javaMM!Field
  mapsTo uml2Property (
    name <- uml2Property.name,
    visibility <- uml2Property.visibility
  )
}

[…]
```

Abb. 5.18 Transformation mittels ATL

Abb. 5.18 zeigt die Anwendung einer solchen Transformationssprache. Als Beispiel wurde die Sprache ATL (ATLAS-Transformation Language) [ATL] gewählt. Im praktischen Teil dieses Buches werden wir noch einen umfassenderen Blick auf eine dedizierte Transformationssprache erhalten, an dieser Stelle daher nur so viel zur Gewinnung eines ersten Eindrucks: Regel-Definitionen werden in Modulen zusammengefasst. Jedes Modul besitzt sowohl ein *Quell-Metamodell* (`IN`), als auch ein *Ziel-Metamodell* (`OUT`). In `from`-Clauses werden deklarativ die Bedingungen an ein Element des Quellmodells beschrieben, die es erfüllen muss, um an der Transformation teilzunehmen. Zu diesen Bedingungen gehören die Übereinstimmung des Element-Typs sowie beliebige weitere Bedingungen an seine Struktur, wie beispielsweise das Vorhandensein bestimmter Stereotypen etc. In der `to`-Clause schließlich stehen die Zielelemente, die erstellt werden, sowie die Aktionen, die nach ihrer Instanziierung auf ihnen erfolgen sollen, um die Transformation abzuschließen.

5.4 Modell-zu-Text Transformationen

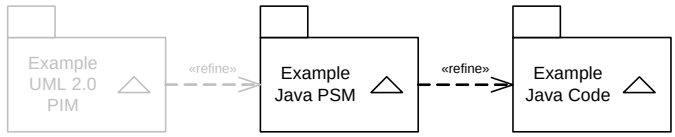
Im vorangegangenen Abschnitt haben wir für die gewählte Beispieltransformation Teile der Programmiersprache Java in einem Metamodell festgehalten und damit implizit die beiden Konzepte der textuellen Sprachsyntax und des Metamodells in Bezug gesetzt. Da

wir bei der nachfolgend zu diskutierenden Modell-zu-Text Transformation (siehe Abb. 5.19) die Grenze zwischen diesen beiden Konzepten überschreiten wollen, müssen wir, bevor wir uns der eigentlichen Erörterung der zugehörigen Technik zuwenden, ihren Zusammenhang näher ergründen.

5.4.1 Fortführung des Beispiels: Java PSM zu Java Code

Bei der Erstellung des Java Metamodells wurde das Thema Sprachspezifikation bereits angeschnitten. Bestandteil einer solchen Spezifikation – also z. B. der bereits erwähnten Java Language Specification – ist die formale Festlegung der Sprachsyntax. Diese Modalität beschränkt sich aber nicht auf höhere Programmiersprachen, sondern gilt für beliebige Sprachen die maschinell ausgewertet werden sollen (siehe 3.1.2).

Abb. 5.19 Vom Java PSM zum Java Code



Dass wir den Bereich der betrachteten Sprachen hier so weit fassen müssen liegt daran, dass die Implementierung eines Softwaresystems in der Regel nicht nur aus dem in einer Programmiersprache verfassten Quellcode besteht.

Abb. 5.20 Ersetzungsregeln in (E)BNF

[…]

MethodDeclaration	::=	MethodHeader MethodBody
MethodHeader	::=	$\text{MethodModifiers}_{opt}$ ResultType MethodDeclarator Throws_{opt}
ResultType	::=	Type `void`
MethodDeclarator	::=	Identifier ($\text{FormalParameterList}_{opt}$)
FormalParameterList	::=	FormalParameter FormalParameterList, FormalParameter
FormalParameter	::=	$\texttt{final}_{opt}$ Type VariableDeclaratorId
VariableDeclaratorId	::=	Identifier VariableDeclaratorId [ ]
MethodModifiers	::=	MethodModifier MethodModifiers MethodModifier

[…]

Vielmehr sind weitere textuelle Artefakte, wie beispielsweise die Beschreibung eines verwendeten Datenbankschemas, z. B. formu-

liert in der Data Definition Language (DDL) [DDL], dazu notwendig, in die wir ebenfalls transformieren bzw. generieren müssen.

Gemein ist solchen Sprachsyntaxen, dass sie oftmals in der *(Erweiterten) Backus-Naur-Form ((E)BNF)* [EBNF] festgehalten sind. Die (E)BNF ist selbst eine Sprache, mit der man die kontextfreie Grammatik einer Sprache durch so genannte Produktionsregeln festhalten kann. Die Produktionsregeln bestehen aus einer linken Seite mit Nichtterminalzeichen und einer rechten Seite mit einer Kombination von Nichtterminalzeichen und Terminalzeichen. Die Regeln definieren, wie die linken Seiten durch die rechten Seiten ersetzt werden können. Abb. 5.20 zeigt beispielhaft einen Auszug solcher Regeln für die Programmiersprache Java.

Die Anwendung der Produktionsregeln erzeugt eine Baumstruktur und wird daher auch Ableitungsbaum oder konkreter Syntaxbaum genannt. In Abb. 5.21 ist exemplarisch der Ableitungsbaum für die Zeichenkette `public String getName()` dargestellt.

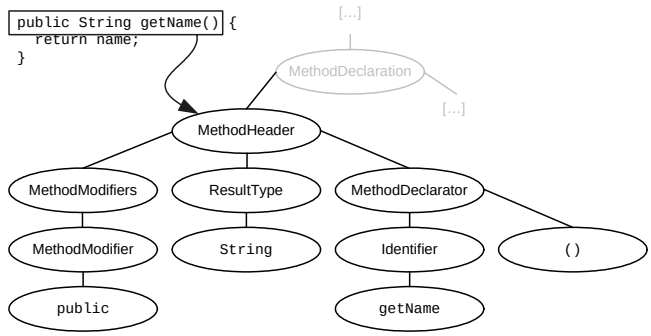


Abb. 5.21 Ableitungsbaum

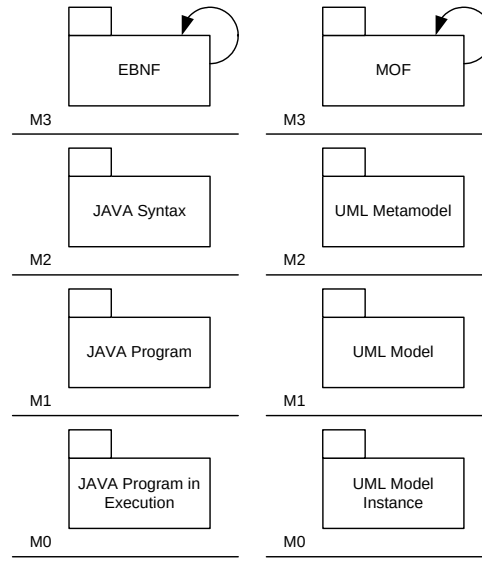
Lässt man die konkreten Sprachsymbole (hier die Blätter des Baums) weg und komprimiert man etwaige Kettenregeln so erhält man den so genannten *abstrakten Syntax Baum (engl. Abstract Syntax Tree (AST))*. Ganz im Sinne unserer Betrachtungen aus Abschnitt 3.1.2 geht dabei die konkrete textuelle Repräsentation verloren und es bleiben die abstrakten Konzepte. Ein solcher AST kann also als In Memory Representation eines Java-Programs – oder allgemein als IMR eines Stück Quellcodes – dienen.

Spannen wir nun den Bogen zum vorangegangen Unterkapitel. Dort hatten wir eine In Memory Representation für Modelle gefunden, die auf der MOF bzw. der UML basiert, und die Form eines Graphen aufweist. Demnach lassen sich die Konzepte der Programmier-

sprache Java in (zumindest) diesen zwei verschiedenen Beschreibungsformen festhalten. Das führt uns zu der Frage, wie sich diese Konzepte in Relation setzen lassen. Betrachten wir dazu die folgende Abb. 5.22.

Die Abbildung skizziert die Vergleichbarkeit der zuvor beschriebenen Konzepte mit denen des Metamodellierungsansatzes. Sie zeigt dazu die Einordnung von EBNF, Syntax, Programm und Programminstanz in die OMG Metamodell Architektur (Metalevel M3-M0) anhand der Programmiersprache Java. Ein Metamodell ist demnach gleichwertig mit der (abstrakten) Syntax einer Sprache zu setzen. Die EBNF nimmt die Rolle der MOF ein, wie diese lässt sie sich ebenfalls mit sich selbst beschreiben.

Abb. 5.22
EBNF vs. MOF



Welchen Erkenntnisgewinn uns die vorhergehenden Ausführungen liefern sollen? Nun, zum einen sollte dies die Einsicht sein, dass wir mit dem, bei unserer Beispieltransformation aus Abschnitt 5.3 eingeschlagenen Weg, ein Quellcode-Modell zu schaffen bereits zu einem wesentlichen Teil die Brücke zwischen Modell und Quellcode geschlagen haben. Die Modell-zu-Modell Transformation von dort vorausgesetzt müssen wir uns also „nur noch“ um die Erzeugung der konkreten Syntax unseres Quellcodes kümmern.

Ein zweiter Punkt wird erst unter Abschnitt 5.4.3 zum Tragen kommen, nämlich dann, wenn wir uns der Synchronisation von Modellen und Code widmen und wir uns dann – gezwungener

Maßen – auch die Rückrichtung, d. h. vom Text-zu-Modell, anschauen müssen.

Aber zurück zur konkreten Syntax. Analog zu den Überlegungen in Abschnitt 5.3.3 werden wir nun eruieren, welche unterschiedlichen Realisierungsmöglichkeiten sich – ausgehend von einer In Memory Representation eines Quellcode- bzw. allgemein eines Text-Modells – für die Umformung in eine textuelle Repräsentation bieten.

5.4.2 … vom Modell zum Text

Auch für den Weg vom Modell zum Text sind im Laufe der Zeit viele Ansätze ersonnen worden, zwei der wichtigsten wollen wir Ihnen nachfolgend vorstellen.

```
package javaMM;

public class Class {
  […]
  private void printFields (PrintWriter pw) {
    for (Field field : getFields()) {
      StringBuffer sb = new StringBuffer();
      for (FieldModifier fm: field.getFieldModifiers()) {
        sb.append(fm.toString() + " ");
      }
      sb.append(field.getType().getName() + " ");
      sb.append(field.getName() + ";");
      pw.println(sb.toString());
    }
  }
  […]
  public void print (PrintWriter pw) {
    printFields(pw);
    printConstructors(pw);
    […]
  }
  […]
}
```

Abb. 5.23 Line-printer

Line-printer

In ähnlicher Weise wie in Abschnitt 5.3.3 mag einem die Verwendung einer populären Programmiersprache auch für die Textgenerierung in den Sinn kommen.

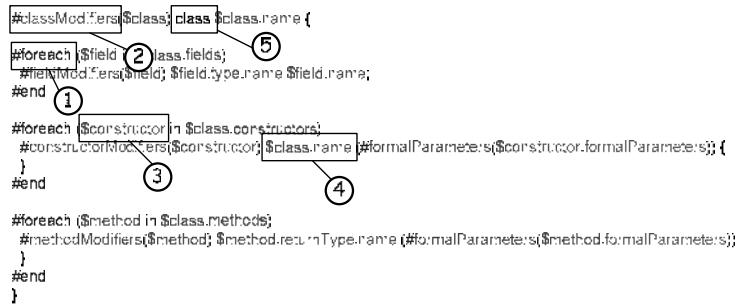
Ein weit verbreitete Herangehensweise basiert beispielsweise darauf, jeden Knoten im Objektgraph zu besuchen und ihn dazu aufzufordern seine textuelle Repräsentation zeilenweise in das zu erzeugende Text-Dokument zu schreiben. Abb. 5.23 zeigt exemplarisch einen Ausschnitt aus einer so gearteten Java-Klasse `javaMM.Class`.

Solche programmierten *„Zeilendrucker“ (engl. line printer)* haben allerdings allesamt den Nachteil, dass man die resultierende Form des zu erzeugenden Text-Dokuments nur schlecht erkennt, da der auszugebende Text quer über das implementierende Model verteilt ist und sie daher schwierig zu warten sind.

Templates

Eine mögliche Antwort auf diese Problematik stellen *Templates (dt. Schablonen)* dar. Templates – oft synonym auch *Boilerplates* genannt – legen das Skelett bzw. den Rahmen des zu erstellenden Textes fest. Um dies zu erreichen, werden statische Textbausteine, die in allen Dokumenten auftauchen, mit Sprach-Elementen der Template-Sprache gemischt, um dynamische Inhalte abzubilden.

Abb. 5.24 Templating mit Velocity



Das nachfolgend vorgestellte *Velocity* ([Velocity] und [GC03]) bietet die Implementierung eines solchen Template-Ansatzes. Velocity bietet als Teil der mitgelieferten *VTL (Velocity Template Language)* neben Makro-Funktionalitäten und der Möglichkeit der Auswertung übergebener Java-Objekte, einen eigenen Kontrollfluss-Befehlssatz, der die Arbeit mit gängigen Datenstrukturen erleichtert.

Abb. 5.24 zeigt Velocity im Einsatz. Dargestellt ist ein Template, das die Generierung des Quellcodes einer einfachen Java-Klasse realisiert. Velocity unterscheidet prinzipiell zwischen Anweisungen (beginnend mit einen „#“), Referenzen (ihnen wird ein „$“ vorangesellt) und einfachem Text.

Referenzen können auf Objekte bzw. Variablen oder aber Methoden verweisen. In letzterem Fall muss die Methode ein entsprechendes Rückgabeobjekt zurückliefern. Objekte werden der Template-Engine aus dem aufrufenden Java-Programm in Form eines Kontextes übergeben. Variablen werden innerhalb eines Templates oder eines Blocks deklariert. In der Abbildung sind beispielhaft einige

Bestandteile eines Templates durch einen Rahmen hervorgehoben und nummeriert:

(1) Als stellvertretendes Beispiel für die Verwendung eingebauter Sprachelemente zur vereinfachten Verarbeitung häufig auftretender Datenstrukturen ist hier die `foreach`-Schleife angeführt. Sie erlaubt die iterative Ausführung eines Blocks von Anweisungen für jedes Element einer übergebenen Menge.

(2) Darüber hinaus erlaubt Velocity die Verwendung benutzerdefinierter Makros. Das angegebene Makro `classModifiers` gibt beispielsweise die Menge der verschiedenen Modifizierer einer Java-Klasse (`public`, `abstract`, `static`, …) als passende Zeichenkette zurück.

(3) Zeigt die Verwendung von Referenzen an, die als Eingabeparameter für Anweisungen oder Makros dienen.

(4) Die Auswertungen dieser Referenzen liefern einen textuellen Rückgabewerte (`String`) zurück der in das resultierende Text-Dokument geschrieben wird.

(5) Der hier eingerahmte Text wird 1-zu-1 in das erzeugte Dokument übernommen.

5.4.3 Synchronisation von Modellen und Code

Eng verzahnt mit der Frage nach Strategien zur Generierung von Code aus Modellen, ist die Suche nach Lösungen zur Synchronhaltung von Modellen und Code während des gesamten Entwicklungszyklus.

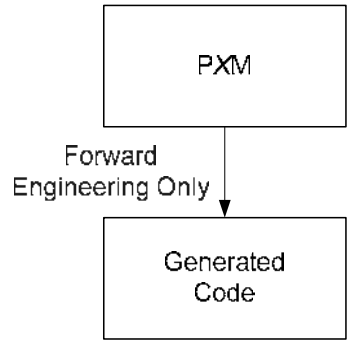
Findet eine Modell-zu-Code Transformation im Projekt erstmalig statt, und startet man sozusagen auf der grünen Wiese, stellt sich die Frage nach Synchronisations-Mechanismen vorerst nicht. Ein Code-Rahmen wird generiert, der vom Entwickler ausgefüllt wird, um zusätzlich benötigte Funktionalität zu integrieren. Dazu benötigt er eventuell zusätzliche Methoden und Felder, die er manuell der generierten Klasse hinzufügt. Gleichzeitig verfeinert der Analyst das Modell, indem er die neuesten Erkenntnisse aus der Analysetätigkeit in das Modell einpflegt. Am Anfang der nächsten Iteration wird dann wiederum Code aus dem verfeinerten Modell erzeugt und … ja, was passiert an dieser Stelle eigentlich? Modell und Code sind parallel weiterentwickelt worden und dadurch auf unterschiedlichem Stand. Wird der geänderte Code einfach überschrieben? Werden Änderungen des einen Artefakts in das andere übernommen und umgekehrt?

Wir wollen uns im Folgenden drei Möglichkeiten anschauen, wie mit der beschriebenen Situation umgegangen werden kann. Die grundsätzlichen Unterscheidungen sind dabei an die Betrachtungen in [Fra03] angelehnt.

Forward Engineering Only

Abb. 5.25 zeigt die erste Möglichkeit mit der eben beschriebenen Situation umzugehen:

Abb. 5.25 Forward Engineering Only



Kurz gesagt findet eine Synchronisation im eigentlichen Sinne schlicht nicht statt. Nicht nur Teile, sondern der gesamte Code wird generiert, und Veränderungen finden nur auf Ebene der Modelle statt. Bei einer Neugenerierung werden alte Code-Dateien komplett überschrieben. Die Executable UML [MB02] fährt beispielsweise einen solchen Ansatz (vgl. Ausführungen in Abschnitt 2.3.2), in der Breite hat sich ein solches Vorgehen allerdings (noch) nicht durchsetzen können. Das liegt sicherlich zum einen daran, dass insbesondere für den Bereich der Verhaltensmodellierung die Ausdruckmöglichkeiten noch unausgereift oder unvollständig sind und die verbleibenden Lücken von Hand programmiert werden müssen (siehe nachfolgenden Abschnitt). Bei den Ausführungen im Praxisteil dieses Buches werden wir noch einmal einen Blick auf diese Problematik werfen und insbesondere die neuen Möglichkeiten der UML 2.0 in Augenschein nehmen.

Partial Round-Trip Engineering

Abb. 5.26 zeigt einen Mechanismus, der die gerade geschilderte Problematik adressiert und eine Vermischung von Modell und Code zulässt.

Es werden also nur Teile der Anwendung generiert, andere müssen manuell von den zuständigen Entwicklern vervollständigt werden. Die Komplikationen, die durch die parallele Veränderung von Modellen und Code auftreten, werden beherrschbar gemacht, indem die manuelle Vervollständigung bestimmten Restriktionen unterliegt, die vom Entwickler befolgt werden müssen.

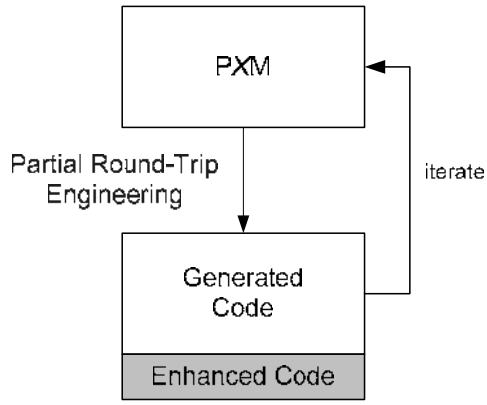


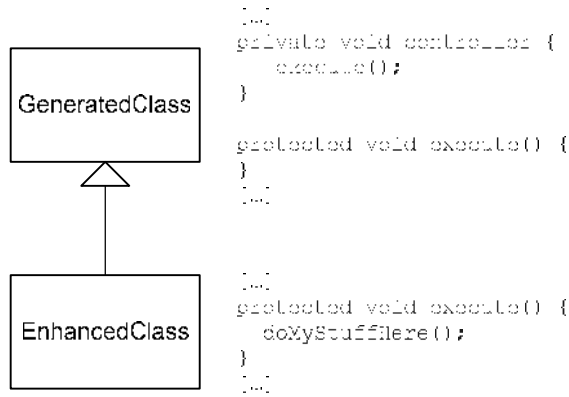
Abb. 5.26 Partial Round-Trip Engineering

Die grundsätzliche Strategie besteht darin, dass lediglich explizite Teile an genau definierten Stellen zum generierten Code hinzugefügt werden dürfen. Veränderungen an generierten Teilen sind verboten bzw. werden bei einer Neugenerierung nicht übernommen. Dies impliziert auch der Name des Ansatzes: Änderungen werden nur teilweise (engl. partial), nämlich nur in Richtung Modell → Code weitergereicht. Manuell hinzugefügte Teile bleiben zwar erhalten, haben jedoch keine Auswirkungen auf die zugrunde liegenden Modelle.

In Abb. 5.27 ist eine Realisierung durch strikte Trennung von generiertem und manuell erzeugtem Code-Teilen dargestellt. Oft wird dies durch die Erzeugung von Basis-Klassen erreicht, von denen dann geerbt werden muss, um sie zu erweitern. Gleichzeitig ist die Einbettung des Kontrollflusses der erbenden Klasse in die übergeordnete Kontrolle eines Frameworks denkbar, das nach dem *Hollywood-Prinzip* [HP] die Methoden der Klasse aufruft. Im Beispiel geschieht dies durch das Überladen einer Methode durch den Entwickler, die dann durch das Framework aufgerufen wird. Grundsätzlich sind jedoch zu diesem Zwecke eine Reihe weiterer Vorgehensweisen denkbar und in Gebrauch. Probleme bei Anwendung dieser Strategie entstehen, wenn Basis-Klassen verschoben, oder umbenannt werden. Hier besteht grundsätzlich die Möglichkeit

der automatischen Anpassung der Benutzer-Klassen durch den Generator, moderne IDEs sind allerdings bereits von Haus aus in der Lage hier Lösungshilfen anzubieten, sodass diese Probleme auch ohne Anpassung des Generators in den Griff zu bekommen sind.

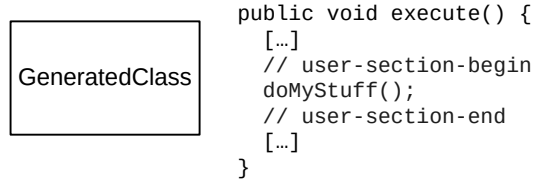
Abb. 5.27 Partial Round-Tripping durch strikte Code-Trennung



In Abb. 5.28 wird eine weitere Möglichkeit gezeigt, die die Vermischung von generierten und nicht-generierten Code-Stücken erlaubt. Einschränkung ist lediglich, dass manuelle Änderungen ausschließlich in *User-Sections* oder auch *Protected Regions* genannten Bereichen stattfinden dürfen. Bei einer Neugenerierung werden diese Sektionen in den neuen Code übernommen, insofern sie nicht vom Entwickler zum Überschreiben freigegeben wurden.

Da User-Sections festen Bereichen innerhalb der umgebenden Klasse zugeordnet werden müssen – Methoden-Körper, Import-Bereich der Klasse, etc. – stellt sich die Frage, was mit diesen passieren soll, falls sich die statische Struktur dieser Klasse grundlegend ändert.

Abb. 5.28 Partial Round Tripping durch geschützte Bereiche



Bei Anwendung dieser Strategie müssen die möglichen Fälle dieser Art im Vorfeld ermittelt, und entsprechende Lösungen im Generator implementiert werden.

Full Round-Trip Engineering

Abschließend zu diesem Themenkomplex noch die dritte Möglichkeit im Umgang mit der Modell-Code Synchronisation (Abb. 5.29).

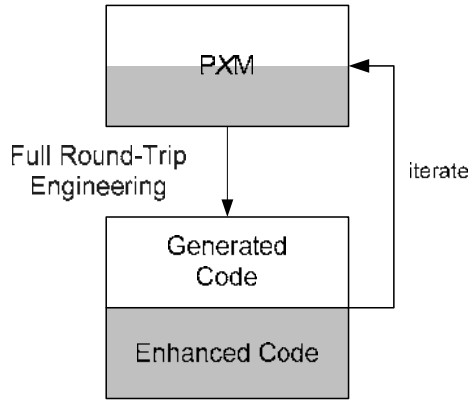


Abb. 5.29
Full Round-Trip Engineering

Änderungen am Code werden auch im entsprechenden Modell wirksam und umgekehrt. Führt der Entwickler beispielsweise im Quellcode ein neues Attribut ein, wird dieses bei der Synchronisation auch im Modell angelegt. Modell und Code sind hier tatsächlich nur unterschiedliche Sichten auf ein und dasselbe System, eine Unterscheidung von generierten und manuell hinzugefügten Teilen findet nicht statt.

Die Meinungen, ob ein solches Full Round-Tripping erstrebenswert ist gehen weit auseinander. Skeptiker befürchten, dass der Vorteil einer Vereinheitlichung der Architektur, die den Systemen durch die zugrunde liegenden Modelle aufgeprägt werden kann, durch die völlige Freiheit der Entwickler diese Architekturprinzipien zu umgehen aufgeweicht wird bzw. völlig verloren gehen kann. Auch die technische Machbarkeit ist bis heute noch nicht bewiesen. Gerade die Synchronisation von Modellen unterschiedlicher Abstraktionsstufen ist ein Thema, dass zurzeit mehr offene Fragen als konkrete Lösungen bereithält.

5.5 PIM → Code vs. PIM → PSM → Code

An dieser Stelle wollen wir noch Betrachtungen anstellen, die unter dem Thema *MDA-light* in Kapitel 2 bereits kurz angerissen worden sind, und sich um die Frage nach der Notwendigkeit plattformspezifischer Modellen drehen. Abb. 5.30 zeigt noch einmal die beiden grundsätzlichen Möglichkeiten.
Während die OMG die Folge PIM → PSM → Code bewirbt, hat sich eine nicht zu vernachlässigende Gruppe von Projekten zusammengefunden, die unter dem Label MDA-light den Wegfall des „unnötigen" Zwischenschrittes PSM postuliert. Hier wird Code direkt und „ohne Umwege" aus dem PIM erzeugt.

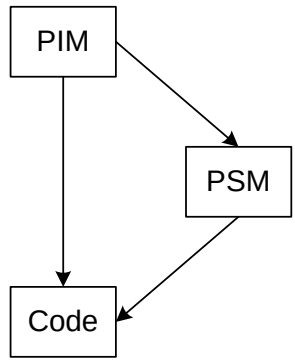


Abb. 5.30 PIM → Code vs. PIM → PSM → Code

Die wesentlichen Argumente für ein solches Vorgehen sind:

- die *fehlende Infrastruktur* in Form von Werkzeugen, die die Realisierung einer solchen Strategie erlauben,
- die dadurch bedingten *hohen Investitionskosten* in Form von eigener Entwicklung und Beratung,
- *MDA-light funktioniert*, dass der volle Ansatz der OMG funktioniert ist noch nicht bewiesen.

Der letzte Punkt ist sicherlich nicht wegzudiskutieren und wir wollen das an dieser Stelle auch gar nicht versuchen. Trotzdem sprechen eine Reihe von Argumenten für die Zwischenschaltung eines PSMs in die Generierungskette:

Wartbarkeit

Transformationen, und die damit verbundenen Transformationsbeschreibungen (/-regeln), von PIM zu PSM und PSM zu Code sind

für sich genommen jeweils weniger komplex als der „lange Weg“ von PIM zu Code. Dadurch resultiert eine höhere Wartbarkeit dieser Regeln.

Modularisierung/Rollentrennung

Bei direkter Generierung von Quellcode aus PIMs werden technische Informationen benötigt, die entweder an das PIM angehängt (durch Stereotypen, Tagged-Values, etc.), oder in der Geschäftslogik des Generators vorgehalten werden müssen. Im ersten Fall wird die Trennung von Technik und Fachlichkeit, und damit einer der Hauptvorteile der MDA geopfert. Sollen Fachspezialisten mit dem Modell arbeiten, wird die Lesbarkeit des Modells durch die „Verunreinigung“ mit technischen Informationen für sie herabgesetzt. Durch die parallele Arbeit von Entwicklern und Analysten am selben Modell werden zudem Synchronisations-Mechanismen sowohl in technischer als auch prozess-gelagerter Sicht notwendig, die bei einer strikten Trennung dieser Aspekte vermieden werden können.

Wiederverwendung von Fachlichkeit

Auch die Anpassung und Wiederverwendung von Fachlichkeit im Falle einer Portierung auf neue Technologien ist nur schwer möglich. Modell und Code-Generierung sind im Allgemeinen so verzahnt, dass eine sinnvolle Wiederverwendung im besten Falle extrem erschwert, und im Worst-Case nahezu unmöglich gemacht wird.

Visuelle Unterstützung der Entwickler

Auch wenn die technischen Informationen im Generator gekapselt werden, und damit bei diesem *Ein-Generator-Pro-Plattform* Ansatz die oben angesprochenen Probleme anscheinend an Relevanz verlieren, lohnt sich unserer Meinung nach die gleichzeitige Generierung eines PSMs – und sei es als *Read-Only Modell* – immer noch. Entwicklern wird die Möglichkeit gegeben, die aus dem PIM generierte Architektur sowie deren unterliegenden Prinzipien und Ansätze, visuell zu erfassen, was der Erfahrung nach leichter geschieht, als durch „scharfes Hingucken“. Dies kann vor allem beim Testen und Debuggen entscheidende Erleichterungen bedeuten. Obwohl PSM und Code konzeptionell auf der selben Abstraktionsstufe stehen, ist das PSM notwendigerweise semantisch reichhaltiger als der resultierende konkrete Code. Eben diese Informationen, wie beispielsweise Vor- und Nachbedingungen, gehen beim Einschlagen des direkten Wegs verloren, und stehen so dem Entwickler in entscheidenden Situationen evtl. nicht zur Verfügung.

Fazit

Alles in allem stellt die „Unterschlagung“ des/der PSM(s) im Entwicklungsprozess also eher ein *Don´t-Do* als eine pragmatische Lösung dar. Die technischen Möglichkeiten (auch im Open-Source Bereich) zur Realisierung der Anforderung werden von Tag zu Tag konkreter und entfernen sich damit von der reinen Vision hin zu durchführbaren Ansätzen. Investitionskosten in Schulung und Entwicklung sind notwendig und nicht wegzudiskutieren. Trotzdem sprechen gewichtige Gründe dafür, sich von den existierenden Schwierigkeiten nicht abschrecken zu lassen. Eine Zwischenlösung wie MDA-light stellt einen guten Anfang dar. Sie sollte jedoch nur als Ausgangspunkt zur Erschließung aller Vorteile des MDA-basierten Vorgehens dienen und nicht das Ende der Entwicklung markieren.

Literatur

[ATL] ATLAS Group: *The ATL home page*.
http://www.sciences.univ-nantes.fr/lina/atl/ (letzter Abruf Mai 2005)

[DDL] Wikipedia: *Data Definition Language*.
http://en.wikipedia.org/wiki/Data_Definition_Language (letzter Abruf Mai 2005)

[EBNF] Wikipedia: *Extended Backus-Naur form*.
http://en.wikipedia.org/wiki/Extended_Backus-Naur_form (letzter Abruf Mai 2005)

[Fow99] Martin Fowler: *Refactoring – Improving The Design Of Existing Code*. Addison-Wesley, 1999.

[Fra03] David S. Frankel: *Model Driven Architecture – Applying MDA To Enterprise Computing*. John Wiley & Sons, 2003.

[GC03] Joseph D. Gradecki und Jim Cole: *Mastering Apache Velocity*. John Wiley & Sons, 2003.

[Groovy] The Codehaus: *Groovy – Home*.
http://groovy.codehaus.org/ (letzter Abruf Mai 2005)

[HP] Portland Pattern Repository's Wiki: *Hollywood Principle*.
http://c2.com/cgi/wiki?HollywoodPrinciple (letzter Abruf Mai 2005)

[JLS] James Gosling, Bill Joy, Guy Steele und Gilad Bracha: *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.
http://java.sun.com/docs/books/jls/download/langspec-2.0.pdf (letzter Abruf Mai 2005)

[JMI] Sun Microsystems: *Java Metadata Interface (JMI)*.
http://java.sun.com/products/jmi/ (letzter Abruf Mai 2005)

[MB02] Stephen J. Mellor und Marc J. Balcer: *Executable UML – A Foundation for Model Driven Architecture*. Addison-Wesley, 2002.

[MOF2] Object Management Group (OMG): *Meta Object Facility (MOF) 2.0 Core Final Adopted Specification*.
http://www.omg.org/docs/ptc/03-10-04.pdf (letzter Abruf Mai 2005)

[MOF2IDL] Object Management Group (OMG): *MOF 2.0 IDL Final Adopted Specification*.
http://www.omg.org/docs/ptc/04-07-01.pdf (letzter Abruf Mai 2005)

[MOF2XMI] Object Management Group (OMG): *MOF 2.0/XMI Mapping Specification, v2.1*.
http://www.omg.org/docs/formal/05-09-01.pdf (letzter Abruf September 2005)

[UML2SS] Object Management Group (OMG): *UML 2.0 Superstructure Final Adopted Specification*.
http://www.omg.org/docs/ptc/03-08-02.pdf (letzter Abruf Mai 2005).

[Velocity] Apache Software Foundation (ASF): *Velocity*.
http://jakarta.apache.org/velocity/ (letzter Abruf Mai 2005)

6 Koordination und Kombination

„But to be really diabolical takes organization, structure, discipline, and method. Taking random potshots and waiting for inspiration with which to victimize the programmer won't do the job. Syntax testing is a primary tool of dirty testing, and method beats sadism every time."

Boris Beizer

Nachdem in den vorangegangenen Kapiteln einzelne Bestandteile der MDA erläutert wurden, wollen wir uns nun anschauen, wie ein auf den Prinzipien der MDA basierender Ansatz im Ganzen praktisch durchgeführt bzw. in ein bestehendes Prozessmodell integriert werden kann. Dabei soll explizit kein ausgearbeitetes Prozessmodell präsentiert, sondern eine Art Rahmen vorgestellt werden, der über bereits bestehende Prozesse gelegt, und so als Hilfsmittel zur Ermittlung und Durchführung eventuell notwendiger Anpassungen dienen kann.

Am Ende sollen die in diesem Kapitel erarbeiteten Ergebnisse als eine Art Baukasten für den eigenen MDA-Prozess dienen. Die Werkzeuge dieses Baukastens bestehen dabei aus:

- Beschreibungen der wichtigsten Aktivitäts- und Workflow-Cluster,
- Änderungen, die am „traditionellen" Rollenmodell der Projektbeteiligten zu erwarten sind,
- Änderungen an der Organisationsstruktur des durchführenden Unternehmens und

- erprobten Prinzipien und Techniken aus dem Software Engineering zur Unterstützung des gesamten Software Entwicklungsprozesses (SEP).

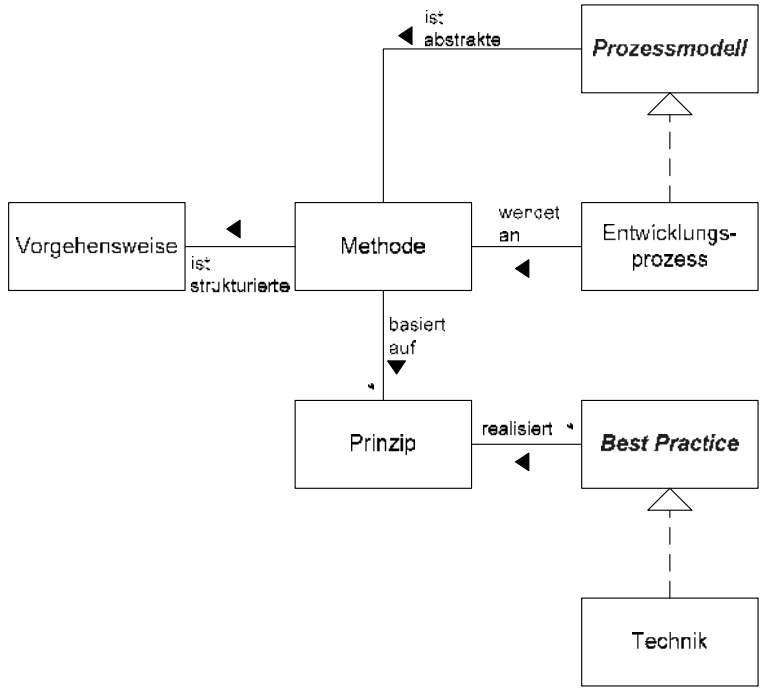
6.1 Grundlagen und Vogelperspektive

Bevor wir ins Detail der angesprochenen Elemente gehen, wollen wir einen Blick aus der Vogelperspektive auf den vorgeschlagenen Prozessablauf wagen sowie Grundlegendes zu dessen Herkunft erläutern.

Terminologie

Da im Bereich der Prozessmodelle zwar reichlich Literatur existiert, jedoch wenige Leitfäden zur Verwendung und Interpretation der benutzten Terminologie zu finden sind, wollen wir an dieser Stelle kurz die in diesem Buch verwendeten Begriffe erklären und einordnen (Abb. 6.1):

Abb. 6.1 Einordnung der Begriffe



Die Beschreibung einer strukturierten Vorgehensweise zur Anwendung von Elementen des Software Engineering wird *Prozessmodell*

genannt. Unter den Begriff fallen sowohl die Beschreibung der durchzuführenden Tätigkeiten, als auch die zu erstellenden Arbeitsergebnisse.

Diese umfassen nicht nur das resultierende Endprodukt, in unserem Falle die Software, sondern auch alle nötigen Zwischenergebnisse wie Dokumentation, Modelle, Projektpläne etc. – die so genannten *Artefakte*.

Entwicklungsprozesse basieren auf Prozessmodellen. Indem sie die Entwicklung eines konkreten Stücks Software den Prinzipien eines Prozessmodells unterwerfen, stellen sie eine Instanz des Modells dar. *Best Practices*, die Erfahrungen aus realen Projekten reflektieren, liefern zusätzlich eine Auswahl erprobter und allgemein anerkannter Prinzipien, die die Wiederholung bereits gemeisterter Probleme verhindern sollen.

Beispiel: Werden in einem Software-Entwicklungsprozess (SEP) Modelle zur Generierung von Quellcode erstellt ist das unterliegende *Prinzip* die Zerlegung eines Problems in Teilaspekte. Die Verwendung von UML 2 zu diesem Zweck stellt eine konkrete *Technik* dar. Da dieses Prinzip der Aspektorientierung ein allgemein anerkanntes, da erfolgreich eingesetztes, Vorgehen darstellt, ist es gleichzeitig *Best Practice*.

Ursprung der Überlegungen

Die im Folgenden angestellten Überlegungen zum Prozessrahmen haben ihren Ursprung in der Welt der Software-Produktlinien und dort vor allem in der in [WeLa99] gegebenen Beschreibung des *FAST*-Entwicklungsprozesses. Uns interessieren an dieser Stelle nur die grundlegenden Prinzipien. Für eine detaillierte Betrachtung des *FAST*-Prozesses selbst verweisen wir auf die genannte Literatur.

6.1.1 Das Prozessmodell von oben

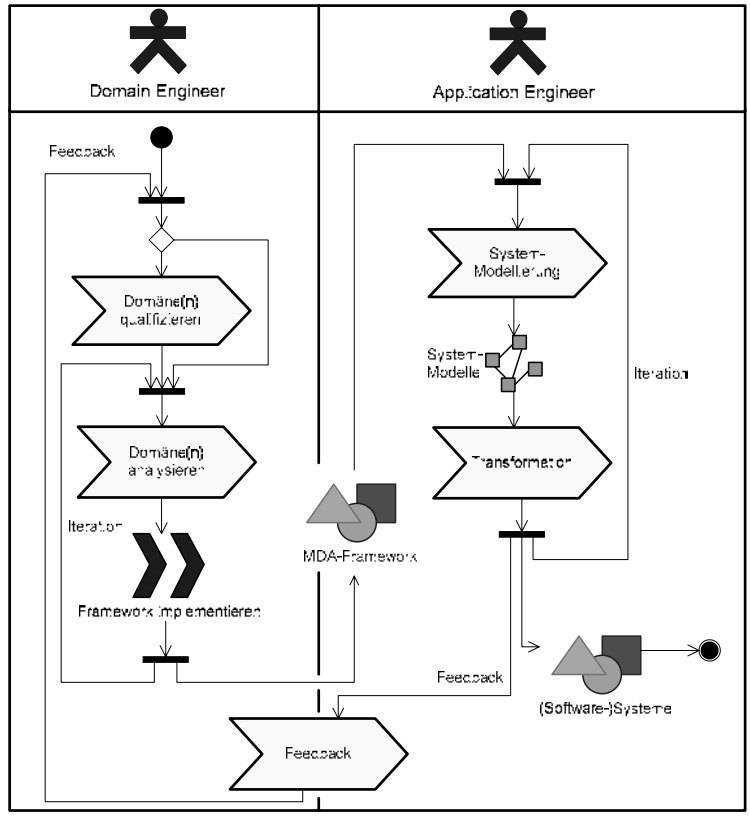
Schaut man aus der Vogelperspektive auf den mithilfe des SPEM UML-Profils [SPEM] modellierten Prozessrahmen werden bereits die grundlegenden Eigenschaften des vorgeschlagenen Vorgehens deutlich (Abb. 6.2).

Dualität der Entwicklungsaufgaben

Vordergründig fällt die essenzielle Zweiteilung der Aktivitäten in die Anwendungsentwicklung (*Application Engineering*) – wahrgenommen durch die Rolle des Application Engineers – sowie des

Domain Engineering und der entsprechenden Rolle des Domain Engineers auf.

Abb. 6.2 Prozessrahmen aus der Vogelperspektive



Die Anzahl der dargestellten Rollen ist zur Vereinfachung der Betrachtung bewusst auf die beiden genannten beschränkt. Damit ein solcher Prozess auch im größeren Rahmen skalieren kann, müssen diese so verfeinert werden, dass anfallende Aufgaben durch Spezialisierung beherrscht werden können. Im Verlaufe des Kapitels (vgl. Abschnitt 6.3) werden wir noch einige spezifischere Rollen näher kennen lernen, vorerst wollen wir jedoch mit unserem groben Überblick fortfahren.

Aufgaben Domain/ Application Engineering

Hauptaufgabe des Domain Engineering ist die Schaffung und Wartung einer technischen und methodischen Infrastruktur zur Anwendung eines MDA-basierten Ansatzes (im Folgenden das *MDA Framework* genannt), während im Application Engineering die Benutzung eben dieses Frameworks zur Erstellung von (Software-) Systemen im Vordergrund steht.

Obwohl in der Darstellung sequenziell angeordnet, können und sollen beide Bereiche der Entwicklung auch parallel betrieben werden. Ob und wie effektiv dies möglich ist, hängt vor allem von der Organisation des durchführenden Unternehmens bzw. der betroffenen Projekte ab. Existiert beispielsweise eine eigene Organisationseinheit, die sich ausschließlich um den Bereich des Domain Engineering kümmert, können beide Bereiche entkoppelt und so weitestgehend parallelisiert werden. Abschnitt 6.5 befasst sich näher mit diesen organisatorischen Aspekten.

Iterativ-Inkrementelle Softwareentwicklung

Lenkt man den Blick auf den zyklischen Ablauf des Prozessmodells, fällt seine zweite grundlegende Eigenschaft auf: die iterativ-inkrementelle Vorgehensweise (IID: Iterative Incremental Development). Diese schlägt sich vor allem in der Art der Arbeit mit den beiden Hauptartefakten, den plattform-unabhängigen und plattform-spezifischen Modellen sowie dem bereits erwähnten MDA-Framework nieder:

Die Systemmodelle spiegeln zu jedem Zeitpunkt den aktuellen Wissenstand des Projektes wider und werden schrittweise zum Endprodukt – der fertigen Anwendung sowie dazugehöriger Dokumentation etc. – weiterentwickelt und verbessert. In der Sprache der MDA bestehen diese Modelle aus allen CIMs, PIMs und PSMs, wobei wir, wie bereits bei der Betrachtung des Modell-Lebenszyklus in Kapitel 4 erläutert, auch Quellcode als sehr spezifisches PSM auffassen wollen.

Die Modelle werden mithilfe der im Framework enthaltenen Werkzeuge erstellt und (ggf. automatisiert) ineinander überführt (transformiert). Geschieht die Modellierung mittels UML, bestehen diese Werkzeuge also aus allen UML-Profilen oder MOF-Metamodellen, den nötigen Mechanismen zur Beschreibung und Durchführung der Transformation von Modellen sowie der nötigen Infrastruktur wie Anleitungen, Prozesse, Testanleitungen usw., die die Durchführung beschreiben und unterstützen.

Lessons Learned

Erfahrungen, die während der Anwendung des Frameworks gemacht werden, fließen am Ende einer Iteration direkt in die Weiterentwicklung des Frameworks und zur Qualitätsverbesserung des Gesamtvorgehens ein. Dieses wesentliche Element des vorgeschlagenen Prozessrahmens erlaubt, gerade bei ersten Gehversuchen mit der MDA, zusammen mit der iterativ-inkrementellen Vorgehensweise ein frühes Lernen aus Fehlern und minimiert so die nicht wegzudiskutierenden Risiken, die der Verwendung eines noch relativ neuen Paradigmas anhaften. Die zum Prozessrahmen gehörende

Aktivität *Feedback* wird in Abschnitt 6.2.6 näher beschrieben. Zu den inhärenten Risiken und Fallstricken kommen wir in Abschnitt 6.6.3.

Diese kurzen Erläuterungen sollen uns vorerst zum prinzipiellen Verständnis des angewendeten Vorgehens genügen. Nähere Erläuterungen sowie konkrete Techniken aus dem Bereich der IID liefert Abschnitt 6.6.1. Nach diesem kurzen Überblick wollen wir nun ins Detail gehen und die beiden Hauptdisziplinen des Prozesses unter die Lupe nehmen.

6.1.2 Domain Engineering

Wie bereits angeklungen beschäftigt sich der Bereich des Domain Engineering innerhalb des vorgeschlagenen SEPs vornehmlich mit allen Aufgaben, die sich um Erstellung, Wartung und Evolution einer unterstützenden Infrastruktur drehen. Definieren wir zunächst einige Grundbegriffe:

Produktlinien, Familien und Domänen

Der Begriff „(Anwendungs-)Domäne“ beschreibt klassischerweise ein Problemumfeld für das eine spezifische Klasse von Systemen erstellt wird. Diese Domänen können grobgranularen Charakter besitzen (Beispiel: Domäne „Versicherungswesen“) oder aber auch sehr feingranular angelegt sein („Logging“, „Programmiersprachen“), wobei grobgranulare Domänen häufig Cluster aus feingranulareren Domänen bilden. So wird die Realisierung des Anwendungsfalls *Partner suchen* mit Sicherheit die Domäne *Persistenz* involvieren, während sich diese wiederum auf Domänen im Umfeld der Programmier- und Abfragesprachen abstützt, usw.

Ziel des Domain Engineering ist es nun, Gemeinsamkeiten und (vorhersagbare) Variabilitäten von (Software-)Systemfamilien in der betrachteten Domäne zu identifizieren, und diese Erkenntnis dazu zu verwenden, Software auf effizientere Art und Weise herzustellen als bisher.

Während im Bereich des SPL-Engineering vornehmlich funktionale Unterschiede in die Analyse der Gemeinsamkeiten und Variabilitäten einbezogen werden – etwa: *Alle Systeme ermöglichen die Intepretation von Konsolenbefehlen zur Laufzeit (Gemeinsamkeit) aber es kann Unterschiede in dem verstandenen Befehlssatz geben (Variabilität)* – spielen in der MDA momentan eher technische Aspekte eine Rolle – beispielsweise: *Alle Programme persistieren*

Daten (Gemeinsamkeit) aber die einen nutzen Entity Beans, die anderen Hibernate (Variabilität).

Dies bedeutet jedoch nicht, dass sich die beiden Ansätze generell auf die Betrachtung einer bestimmten Granularitätsebene beschränken. So bietet beispielsweise die MDA Möglichkeiten, sowohl technische als auch funktionelle und fachliche Aspekte in einem modellgetriebenen Ansatz zu vereinen. Ein Beispiel zur Modellierung funktioneller Variabilität haben wir bereits in Kapitel 4 mit dem Beispiel des SimplePLE Profils kennen gelernt. Die Modellierung fachlicher Aspekte, die dem Kontext entsprechen, in dem die zur Verfügung gestellte Funktionalität zu den umgebenden Geschäftsprozesse steht, wird unter anderem durch die OOGPM [OWS+03] ermöglicht. Mit dieser Methodik zur Beschreibung von Geschäftsprozessen mittels UML 2.x werden wir uns im weiteren Verlaufe dieses Buches noch näher beschäftigen.

Der Unterschied des Domänenbegriffs der MDA zur „klassischen" Definition innerhalb der Welt der SPL zeigt sich also vor allem in der Granularität der Domänen, die in der Betrachtung eine Rolle spielen. Während in SPLs möglichst viele Mitglieder einer Familie gebaut werden sollen, gehören Systeme, die mittels MDA erstellt werden tendenziell einer Vielzahl von Familien an; Beispiel MDA: *Eine Anwendung, die Daten persistiert, Transaktionen benötigt, eine Web-Oberfläche zur Verfügung stellt und den Geschäftsprozess der Angebotserstellung unterstützt.* Die Domänen, denen eine solche Anwendung angehören würde, sind also: Persistenz, Transaktionen, Client-Interaktion und Geschäftsprozesse.

Wir haben diesen Gedanken bereits an früherer Stelle dieses Buches (in Kapitel 4) unter etwas anderer Terminologie kennen gelernt. Grobgranulare Domänen wurden dort als vertikale, die eben als feingranular-technisch charakterisierten, als horizontale Domänen bezeichnet. Im selben Kapitel haben wir ebenso anhand von Beispielen gesehen, dass die MDA grundsätzlich zur Arbeit sowohl mit horizontalen als auch mit vertikalen Domänen geeignet ist. Die Adaption des Prozessmodells aus der Welt der SPLs auf die Welt der MDA sollte daher kein größeres Problem darstellen.

Interessante Beobachtung an dieser Stelle ist, dass auch der verwendete Entwicklungsprozess selbst eine Domäne in oben angeführtem Sinne darstellt. So sollte jede gute, mittels MDA-Prinzipien hergestellte Software in der Domäne der Anwendungen liegen, die mithilfe des besagten iterativ-inkrementellen Prozessrahmens realisiert

werden. Wie dies in einem Unternehmen konkret umgesetzt wird, also die konkrete Instanz des IID-Prozesses, ist individuell unterschiedlich (variabel). Der Schritt von der abstrakten Idee eines solchen Rahmens hin zu einem kompletten, funktionierenden und auf die unterschiedlichen Anforderungen einzelner Projekte zugeschnittenen Prozessmodell gehört zum wertvollen Fachwissen des Unternehmens. Tatsächlich gehen einige MDA-Methoden so weit das Prozessmodell komplett in die Modellkette der einzubeziehen und durch Anpassung eines vorgegebenen Referenzmodells auf die individuellen Bedürfnisse des aktuellen Projektes den tatsächlich zur Anwendung kommenden Prozess zu „generieren".

Welche Domänen kommen in Frage?

Die letztendliche Entscheidung, ob eine Domäne für eine Verwendung innerhalb des eigenen Prozesses vom finanziellen Standpunkt aus in Frage kommt oder nicht, liefert die Aktivität *Domäne qualifizieren* (Abschnitt 6.2.1). Die ursprünglich angeklungene Frage, welche Domänen sich überhaupt für die Verwendung in der MDA eignen ist nicht für die MDA im Allgemeinen zu beantworten, sondern kann nur durch eine genaue Untersuchung der im eigenen Haus durchgeführten Software-Projekte geliefert werden. Dabei werden sowohl bereits abgeschlossene Projekte sowie die dort erstellten Systeme, als auch evtl. geplante Software auf verwertbare Gemeinsamkeiten untersucht. Man spricht bei einer solchen Untersuchung von einer *Gemeinsamkeiten-Variabilitäten Analyse (engl. Commonality Variability Analysis).* Wie eine solche Analyse aussehen kann, wird in [WeLa99] ausführlich an Fallstudien erläutert.

Ziel des Domain Engineering

In der Summe ist das Ziel des Domain Engineering also die Erstellung und Wartung des verwendeten MDA-Frameworks. Die Bestandteile des Frameworks gliedern sich prinzipiell in zwei Gruppen:

- *den wiederverwendbaren Softwarebestand (Sprachen, Modelle, Architekturen, Muster, vorgefertigte Softwarekomponenten, Werkzeuge (z. B. Transformatoren/Generatoren), …)* zur Erstellung von Anwendugen sowie
- *unterstützende Prozesse* und organisatorische Aspekte zur Durchführung von Software- und Framework-Entwicklung

Wir wollen die Bestandteile im Folgenden die *Assets* des Frameworks nennen.

6.1.3 Application Engineering

Werden im Domain Engineering die Voraussetzungen zur Durchführung von MDA-Projekten bereitgestellt, ist es Aufgabe des Application Engineering diese zu nutzen, um konkrete Anwendungen zu produzieren. Der dazu benutzte Entwicklungsprozess stammt dabei ebenfalls aus dem Fundus des Frameworks und unterscheidet sich vom grundsätzlichen Ablauf meist nicht von herkömmlichen iterativ-inkrementellen Vorgehensmodellen (vgl. dazu auch Abschnitt 6.1.1).

Anforderungen werden aufgenommen und mithilfe der DSLs des Frameworks festgehalten. Es folgt das Design, das allerdings, im Gegensatz zum traditionellen Vorgehen, nicht nur durch die Evolution, sondern zusätzlich teilweise oder vollautomatisch durch die Transformation bestehender Modelle stattfindet. Die letztendliche Generierung der Anwendung, in Teilen oder vollständig, findet wieder mit Mitteln des MDA Frameworks statt.

6.2 Aktivitäten und Artefakte

Wir wollen uns nun die Aktivitäten, die innerhalb des vorgestellten Prozessrahmens (Abb. 6.2) stattfinden, näher anschauen. Auch hier geht es darum den „Kern“ der durchzuführenden Aufgaben zu erfassen und nicht um eine Beschreibung einer vollständigen Methodik. Auch hier sind die anstehenden Aufgaben weiter aufzugliedern und feingranularer zu beschreiben, wenn das Modell in größeren Projekten skalieren soll.

6.2.1 Domäne qualifizieren

Bevor eine Domäne in das MDA-Framework aufgenommen wird, steht die Untersuchung, ob sich eine solche Aufnahme finanziell lohnt. Dazu müssen anfallende Kosten mit und ohne MDA-Ansatz geschätzt und diese in einem Modell gegenübergestellt werden, um eine begründete Entscheidung zu ermöglichen.

Economic Model

Wir wollen diese Aktivität als das *Qualifizieren* einer Domäne bezeichnen und das angesprochene resultierende Endprodukt der Qualifikation das *Economic Model* nennen.

Vorüberlegungen

Wichtigste Größe bei den anzustellenden Überlegungen sind die Gesamtkosten, die pro Projekt für Aktivitäten im Bereich der betrachteten Domäne auflaufen. Diese werden sowohl für den aktuell verwendeten Ansatz als auch näherungsweise für die angestrebte

MDA basierte Vorgehensweise mit angenommenem oder bereits existierendem Framework ermittelt.

Da die Effektivität von MDA stark von dem Konzept und der Verwirklichung des Wiederverwendungsgedankens abhängt, muss auch dieser in unseren Überlegungen berücksichtigt werden. Wir wollen deshalb zusätzlich die zeitliche Entwicklung der Kosten der Gesamtzahl der Projekte gegenüberstellen, die mit dem bestehenden bzw. neu geplanten Vorgehen durchgeführt worden sind.

Notwendige Schritte

Um die angeführte Kostengegenüberstellung bewerkstelligen zu können, ist zuerst die Erstellung einer Liste von Aktivitäten erforderlich, die im Zusammenhang mit der betrachteten Domäne in einem Projekt durchzuführen sind. Die zum Vergleich benötigten Gesamtkosten ergeben sich dann aus der Summe der Kosten pro angeführter Aktivität. Diese Betrachtung ist sowohl für das herkömmliche Vorgehen als auch für die (geplanten) MDA-Projekte durchzuführen.

Beide Betrachtungen können dabei nur Schätzwerte liefern. Ein Ansatz, um zu vergleichbaren Ergebnissen zu kommen, ist die Vorstellung eines imaginären „Standard-“ bzw. „Referenzprojektes“, das in Größe und Komplexität einem Durchschnitt der bislang im Hause durchgeführten Softwareprojekte entspricht und zur Berechnung der Kosten herangezogen wird.

Eine Besonderheit für die Ermittlung der Kosten des MDA-Ansatzes ist die Schätzung der initial auftretenden, einmaligen Investitionen, die zur Integration der Domäne in den SEP zu tätigen sind. Hierzu zählen Kosten für Ausbildung, Anschaffungen, Umstrukturierungen, Gewöhnungsphase etc., die zur Umsetzung des Ziels getätigt werden müssen. Auch hier kann nur die Erfahrung des Unternehmens als Hilfsmittel zur Begrenzung der gegebenen Ungenauigkeit dienen. Bester Rahmen zur Durchführung der Schätzung ist daher ein Workshop mit Vertretern aller beteiligten Organisationseinheiten sowie des geldgebenden Managements.

Don't underestimate

Typischerweise werden in diesem Segment anfallende Kosten chronisch unterschätzt. Die Best Practice *„Technologien realistisch einschätzen“* (Abschnitt 6.6.2) gibt Tipps, um diese Klippe zu umschiffen.

Hat man alle Daten ermittelt, können diese in einem *Business-Case* gegenübergestellt und auch grafisch dargestellt werden.
Abb. 6.3 zeigt ein Beispiel für eine solche Darstellung.

Im Ursprung startend, als durchgezogene Linie dargestellt, die Gesamtkosten einer Softwareentwicklung bei Durchführung eines herkömmlichen Ansatzes. Unser imaginäres Standardprojekt hat konstante Kosten in der betrachteten Domäne, weshalb sich im Verlauf über die Anzahl der produzierten Software-Systeme in der Zeit eine Gerade ergibt. Die flankierenden gestrichelten Linien stellen einen Korridor dar, der die Unsicherheit der Schätzung symbolisieren soll.

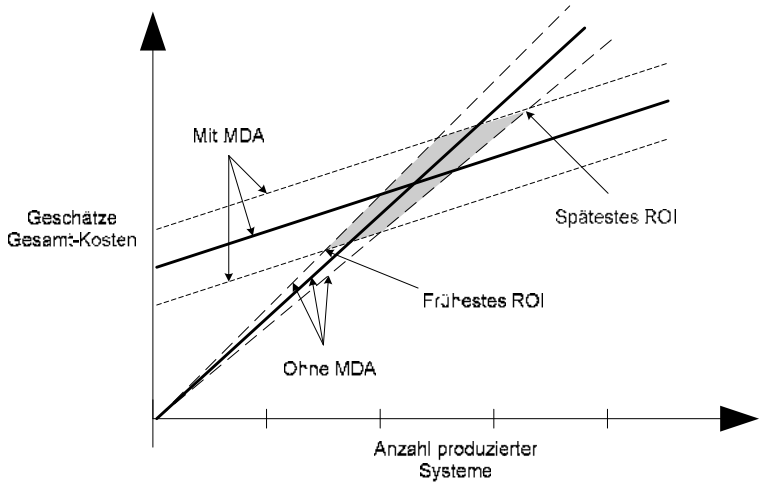


Abb. 6.3 Grafische Darstellung Economic Model

Die Kosten desselben Standardprojektes unter MDA-Bedingungen durchgeführt, startet mit initialen Investitionskosten, die bereits vor der bzw. parallel zur Erstellung des ersten Systems zu tätigen sind. Die konstanten Kosten pro Projekt sind im Idealfall allerdings geringer als in herkömmlichen Projekten, weshalb der Schnittpunkt der Geraden den erhofften *Return-on-Investment (ROI)* darstellt.

Ergebnis der Aktivität

Output der durchgeführten Aktivität ist das Economic Model der betrachteten Domäne. Dieses besteht aus den erstellten Aktivitätslisten nebst zugehöriger Kostenschätzungen sowie der Gegenüberstellung in einem geeigneten Business-Case, der eventuell die oben angeführte graphische Auswertung enthalten kann. Eine Domäne ist dann als qualifiziert zu betrachten, wenn die Kosten eines Standard-Projektes, das in der Domäne liegt, mit MDA niedriger liegen als in der herkömmlichen Produktionsweise, was sich im graphischen Modell durch einen geringeren Anstieg der Geraden bemerkbar macht. Welche Differenz der anfallenden Kosten pro Ansatz noch als akzeptabel gilt, ist vor allem eine Frage des finanziellen Atems.

Je niedriger die Differenz, desto länger die Zeit bis zum Break-even. Hier zahlt sich geleistete Motivationsarbeit bei den geldgebenden Instanzen im Vorfeld aus, was die Betonung der vorigen Abschnitte auf der möglichst frühen Integration von Vertretern des geldgebenden Managements bereits in den ersten Phasen der Analysen erklärt.

Auch strategische Entscheidungen und das Gewicht der Domäne – sprich Häufigkeit des Auftretens etc. – spielen bei der letztendlichen Entscheidung eine Rolle. Diese sind jedoch so individuell unternehmensspezifisch, dass wir an dieser Stelle nicht näher auf das Thema eingehen können. Zur ausführlichen Betrachtung dieses Themengebietes empfehlen wir die Lektüre von [CN01], die einen ausführlichen Überblick über die verschiedenen Aspekte dieses Themenkomplexes bietet.

6.2.2 Domäne analysieren

In dieser Phase des Vorgehens geht es darum die in Abschnitt 6.2.1 als qualifiziert befundenen Domänen für die Verwendung in einem modellgetriebenen Ansatz, insbesondere der MDA, nutzbar zu machen. Dafür sind vor allem geeignete domänen-spezifische Sprachen zu entwickeln, um Aspekte der Domäne modellieren zu können. Außerdem sind die Spezifikationen der benötigten unterstützenden Software zur Generierung und Transformation zu erstellen, aber auch eventuell notwendig werdende Anpassungen von Prozessen und Organisationen innerhalb des Unternehmens zu beschreiben und deren Umsetzung voranzutreiben.

Wir wollen uns an dieser Stelle auf das Themengebiet der Modellierungs- und Transformationssprachen beschränken. Abschnitt 6.3 liefert Gedanken zu evtl. Anpassungen von Projektaufgaben, Abschnitt 6.5 geht auf organisatorische Aspekte ein. Die Entwicklung der notwendigen Werkzeug-Infrastruktur (Transformator, Generator, …) sowie die dazu nötige Spezifikation der Werkzeug-Funktionalität wird im Fallbeispiel des Buchs durchgeführt und soll an dieser Stelle als bereits existierend vorausgesetzt werden.

Vorüberlegungen

Die Wiederverwendung von Fachwissen in der MDA beruht auf der Hoffnung, dass Domänenwissen auf einer allen Mitgliedern der Systemfamilie gemeinsamen Ebene abstrahiert werden kann und konkrete Individuallösungen variierte Ausprägungen dieses abstrahierten Fachwissens darstellen. So ist das Wissen darum, dass eine Klasse `Konto` persistenten Charakter hat, allgemein wiederverwendbares Fachwissen. Die Individuallösung „`Konto` ist als EJB

EntityBean realisiert“ ist somit eine systemtechnische Realisierung dieses Fachwissens. Mit JavaEE/EJB als Plattform ist die erstaufgeführte Beschreibung plattform-unabhängig, letztere plattformabhängig. Der Übergang zwischen beiden – übrigens auch kostbares Fachwissen – wird in einer Transformationsvorschrift festgehalten.

Notwendige Schritte

Zu erstellen sind plattform-(un)abhängige Sprachen zur Beschreibung von Domänen unterschiedlichen Abstraktionsniveaus sowie die Transformationsvorschriften zur Überbrückung der einzelnen Abstraktionsstufen. Im Falle der Hauptmodellierungssprache UML besteht die Hauptaufgabe also in der Erstellung von Profilen für CIMs, PIMs und PSMs.

Abstraktion: Bottom-up

Zur Erstellung eines UML-Profils bzw. UML-Metamodells, das die domänen-spezifischen Aspekte modellierbar macht, sind zwei grundsätzliche Vorgehen denkbar. Wird erstmals ein Modell der Domäne erstellt und existieren wenig Erfahrungen im Metamodellierungsprozess, ist ein *Bottom-up* Vorgehen sinnvoll. Dabei entscheidet man sich zuerst für eine Technologie (=Plattform), die im Projekt zur Realisierung der Domänen-Aspekte verwendet werden soll. Erst danach werden mithilfe von Experten in einem Workshop die Kernkonzepte dieser Technologie ermittelt und mittels MOF oder UML Infrastructure ein Profil bzw. Metamodell für diese Plattform erstellt. Von diesem PSM-Profil ausgehend wird dann versucht das PIM-Profil der Domäne zu extrahieren, indem alle technologie-spezifischen Elemente „wegabstrahiert“ werden und nur noch die Essenz der Domäne erfasst wird. Der Weg, um von Instanzen des PIMs wieder zu PSMs zu gelangen entspricht dann dem genau umgekehrten Weg, in dem das PIM wieder mit den zuvor abstrahierten Details angereichert wird. Genau das Wissen darum, welche Details dies sind und unter welchen Bedingungen sie hinzugefügt werden können ist dann in der Transformationsbeschreibungssprache des Frameworks festzuhalten.

Abstraktion: Top-down

Existiert bereits ein PIM und soll eine neue technologische Ausprägung dem Framework hinzugefügt werden (PSM), oder existiert vielleicht gar nicht die Möglich- oder Notwendigkeit eines PSMs, da mithilfe eines CIMs Geschäftsprozesse o. Ä. festgehalten werden sollen, bietet sich ein *Top-down* Vorgehen an. Auch hier müssen die Kernkonzepte der Domäne mithilfe von Domänen-Experten ermittelt und durch den verantwortlichen Architekten in einem Metamodell oder korrespondierendem Profil festgehalten werden. Der Unterschied zum vorangegangenen Ansatz liegt lediglich im Start-

punkt der Untersuchungen. Der erste Ansatz beginnt mit der plattform-spezifischen Technologie, während der zweite Ansatz den initialen Fokus zuerst auf den Kontext der Problem- bzw. Fachdomäne legt.

Die Beschreibung des eigentlichen Vorgangs der Abstraktion geschieht notwendigerweise sehr vage. Das Erkennen und „Herauspräparieren“ der Kernkonzepte von Technologien und Domänen kann nur aus gelebter Erfahrung in der Domäne heraus geschehen und verlangt die Zusammenarbeit der Architekten des Frameworks mit entsprechenden Domänen- und Technologie-Experten.

Heuristiken als Empfehlung

In [LM97] versuchen die Autoren, heuristische Tipps zum Vorgehen zu liefern und grundsätzliche Techniken zur Durchführung einer Domänenanalyse zu identifizieren. Ob dies gelungen ist, mag der Leser selbst entscheiden. Die Lektüre des Papers ist auf jeden Fall zu empfehlen.

Das Rad nicht neu erfinden

Wo immer möglich, sollten Standard-Profile genutzt werden, um das Rad nicht unnötigerweise neu zu erfinden. Das Studium bereits existierender Profile als Startpunkt empfehlen wir aber auf jeden Fall, auch wenn eine Eigenentwicklung unvermeidlich sein sollte. Auf den Seiten der OMG [UMLProfiles] finden sich bereits einige Profile zu den unterschiedlichsten vertikalen und horizontalen Domänen, aber auch die Benutzung von Suchmaschinen liefern bereits einige Fundstücke, die als Vorlage und Beispiel für selbstentwickelte Profile dienen können.

Ergebnis der Aktivität

Ergebnisse der Domänen-Analyse sind die DSLs, die zur Modellierung der domäneneigenen Aspekte benötigt werden – also beispielsweise Metamodelle bzw. Profile für CIMs, PIMs u. PSMs – aber auch die Spezifikationen für benötigte Werkzeuge, Prozess-/Organisationsanpassungen, sowie die Transformationsregeln zum Übergang von plattform-unabhängigen Modellen zu entsprechenden PSMs. Während die Modellierungssprachen direkt als Input für die Aktivitäten des Application Engineering dienen, gehen die Spezifikationen als Grundlage der Arbeit in die vorerst letzte Stufe des Domain Engineering ein: der Realisierung des Frameworks.

6.2.3 Framework implementieren

Die Implementierung des Frameworks entspricht in seinen Phasen und grundsätzlichen Vorgehensweisen denen eines Standard SEPs mit anschließender während der gesamten Lebensdauer des Frame-

works durchgeführter Feedback-Schleife zur kontinuierlichen Optimierung und Evolution der enthaltenen Bausteine.

Die methodischen Besonderheiten dieser Aufgabe entsprechen denen, die wir im Verlaufe dieses Kapitels kennen gelernt haben, bzw. noch kennen lernen werden. Auf die technischen Aufgaben und die Angabe einer prototypischen Lösung geht die Beschreibung des Fallbeispiel im zweiten Teil des Buches ein und liefert so einen zusätzlichen Eindruck von den anstehenden Aufgaben innerhalb eines solchen Prozesses. Die Beschreibung der Feedback-Aktivität übernimmt Abschnitt 6.2.6.

6.2.4 System-Modellierung

Die Aktivität *System-Modellierung* im MDA-Kontext entspricht verallgemeinernd gesprochen der Menge von Aktivitäten herkömmlicher Softwareprozesse, die sich mit der Erfassung und Modellierung von Anforderungen, ihrer Analyse und dem System-Entwurf befassen. Die Methoden zur Erfassung unterscheiden sich bei beiden Ansätzen kaum (vgl. dazu Abschnitt 6.3). Die Besonderheit innerhalb MDA sind die Mittel, mit denen die Ergebnisse der Bemühungen festgehalten werden. Ergebnisse, die als qualifiziert befundene Domänen betreffen, werden mithilfe der erstellten DSLs modelliert. Zusätzliche Besonderheit ist die Tatsache, dass Modelle nicht vollständig auf der grünen Wiese entstehen, sondern mithilfe von Transformatoren automatisiert ineinander überführt bzw. mit Details angereichert werden. In der vorliegenden Aktivität wird der manuelle Vorgang der Erstellung und iterativen Überarbeitung der Modelle betrachtet. Auf den Vorgang der Transformation gehen wir im folgenden Abschnitt (Abschnitt 6.2.5) ein.

Notwendige Schritte

Prinzipiell sind zwei wesentliche Unteraktivitäten durchzuführen. Die Erste beschäftigt sich mit der Neuerstellung von Modellen auf Basis der auf herkömmliche Weise ermittelten funktionalen und nichtfunktionalen Anforderungen. Dies können Modelle von Geschäftsprozessen sein (typischerweise CIMs) oder Modelle, die bereits die technische Unterstützung dieser Prozesse mit den entsprechenden plattform-unabhängigen Profilen in Betracht ziehen.

Abstraction Gap

Zweite Hauptaktivität ist die Verfeinerung von Modellen, die durch Transformationen aus den eben beschriebenen Modellen hervorgegangen sind. Die automatische Transformation kann in der Regel nicht alle Lücken zwischen den Abstraktionsebenen der beteiligten PIMs und PSMs stopfen. Hier muss in Handarbeit das aus der

Transformation hervorgegangene Modell mit den benötigten Zusatzinformationen versorgt werden. Typischerweise sind hier vom Entwickler Einstellungen für das „Feintuning“ der verwendeten Zieltechnologien zu setzen sowie – so zumindest der Status quo – die notwendige Fachlogik manuell zu codieren.

Verhaltensmodellierung

Mit der Präzisierung der UML in der aktuell vorliegenden Version haben sich allerdings neue Möglichkeiten ergeben von der reinen Generierung struktureller Aspekte wegzukommen. So werden beispielsweise Aktivitätsdiagramme in zunehmendem Maße dazu benutzt auch Verhaltensaspekte im Modell darzustellen und so die Möglichkeit geschaffen die Lücke zwischen Modell und Code (*engl. model-code gap*) weiter zu verkleinern. Im weiteren Verlauf des Buches werden wir konkrete Beispiele für die Verhaltensmodellierung und der daraus resultierenden Erzeugung dynamischer Anwendungsteile kennen lernen.

Ergebnis der Aktivität

Die Menge aller nicht durch Transformation hervorgegangener sowie der manuell überarbeiteten Modelle stellt das Ergebnis der Aktivität dar. Also alle CIMs, PIMs und PSMs. Vergessen werden sollte nicht, dass textuelle Spezifikationen und sonstige Artefakte, die aus der Projektarbeit hervorgegangen sind, auch Modelle darstellen und ebenfalls unter diese Betrachtung fallen.

6.2.5 Transformation

Die Überbrückung der Abstraktion von plattform-unabhängigen Modellen hin zur letztendlichen Realisierung mittels geeigneter Technologien bietet die Transformation.

Notwendige Schritte

Die Durchführung der Transformation, worunter wir auch die Code-Generierung als Sonderform verstehen wollen, ist stark von den vorhandenen Werkzeugen abhängig. Die theoretischen Grundlagen dazu haben wir im vorangegangen Kapitel gelegt. Wie die konkrete Manifestation einer Transformation aussieht, wird an späterer Stelle noch demonstriert. Prinzipiell bleibt an dieser Stelle festzuhalten, dass dazu ein Quellmodell und die zugehörigen Transformationsvorschriften auf denen der Transformator arbeiten soll anzugeben sind. Die Spezifikation der genannten Transformationsvorschriften selbst geschieht bereits in der Domänen-Analyse (Abschnitt 6.2.2).

One-Step Build-Prozess

Für die Durchführung der Transformation ist es dabei wünschenswert, einen One-Step Build-Prozess (vgl. Abschnitt 6.6.3 *Phlegmatischer Build-Prozess*) zu haben, der sich direkt aus der Entwicklungsumgebung starten lässt. Dieser Prozess reicht vom (möglicherweise notwendigen) Modell-Export, über die angesprochene Transformation bzw. Generierung bis hin zur Compilierung, Paketierung etc. Probleme, neue Anforderungen oder Anforderungsänderungen, die sich aus der Durchführung dieses Prozess ergeben, werden gesammelt und fließen in die Aktivität Feedback (siehe 6.2.6) ein.

Ergebnis der Aktivität

Ergebnis der Transformations- und Generierungsaktivität sind die erstellten Modelle und generierte Systemartefakte wie Deskriptore und Code-Rahmen. Sie dienen zur Verfeinerung als Input der Modellierungs-Aktivität (vgl. Abschnitt 6.2.4) oder werden als ausführbares Release an den Kunden ausgeliefert.

6.2.6 Feedback

In der Feedback genannten Aktivität am Ende einer Iteration findet eine Bewertung des MDA-Frameworks gemessen an den Ansprüchen der alltäglichen Entwicklung innerhalb des Application Engineering statt. Die Aufgaben, die innerhalb dieser Aktivität zu erledigen sind, betreffen sowohl die erstellte Kundenanwendung (als Ergebnis des Application Engineerings) als auch die Ermittlung von Handlungsbedarf im Domain Engineering, der sich aus Problemen und Optimierungsvorschlägen ergibt, die während der Anwendungserstellung aufgetaucht sind. Grundsätzlich stehen folgenden Aufgaben an:

- Die Begutachtung des Softwareproduktes durch den Kunden,
- die Bewertung der Zielerreichung,
- die Bewertung der Code-Qualität,
- die Auswertung der Testergebnisse (funktional/nichtfunktional),
- die Diskussion der verwendeten Methodik sowie aufgefallener Punkte innerhalb des MDA-Frameworks mit dem gesamten Team und schließlich
- die Planung bzw. Planungskorrektur für die nächsten Projektabschnitte.

Die Ergebnisse dieser Aktivitäten werden analysiert und die daraus resultierenden Aufgabenpakete an die beiden Entwicklungsbereiche weitergereicht. Änderungen, die sich aus den Ergebnissen des letzten Feedbacks ergeben haben, werden am Anfang des nächsten Projektabschnitts dem gesamten Team mitgeteilt. Beispiele für solche Änderungen können sein:

- Die Anpassung der Abschnittsziele/-prioritäten,
- die Änderung der Anforderungen durch den Kunden,
- Änderung an der Architektur/Infrastruktur des MDA-Frameworks (der dort enthaltenen Tools, usw.) sowie
- die Anpassung der verwendeten Methodik

Wie man sieht, hat das Feedback direkte und u. U. drastische Auswirkungen auf das gesamte weitere Vorgehen innerhalb des Projektes. Typischerweise wird jedoch nicht das komplette bisherige Vorgehen über den Haufen geworfen, sondern Änderungen behutsam und schrittweise eingeführt.

Notwendige Schritte

In allen Aktivitäten des Application Engineering müssen sich die Gedankenkonstrukte und Werkzeuge, die in den Köpfen und Editoren der Domain Engineers gewachsen sind, an den Anforderungen der tatsächlichen „harten“ Softwareentwicklung messen. Da diese Entwicklung vollständig auf Assets des MDA-Frameworks aufbaut, müssen am Ende einer Iteration auch alle Assets zur Diskussion stehen dürfen, die sich als nicht alltagstauglich erwiesen haben und damit angepasst, neu überdacht oder gar verworfen werden müssen.

Typische Kandidaten, die einer Überprüfung unterzogen werden sind:

- die zur Modellierung verwendeten Notationen (Profile und Metamodelle),
- die benutzten Transformatoren und Generatoren,
- die Transformationsvorschriften und zur Generierung benutzten Templates,
- die verwendeten Verfahrensweisen und Techniken,

mit anderen Worten: alle Hilfsmittel, die von der Erstellung der Modelle bis zur letztendlichen Generierung und Auslieferung der Anwendung benötigt werden.

Um diese Aufgabe zu erleichtern, ist es unter Umständen sinnvoll ein *Problem-Tagebuch* zu führen und diesen Task fest in das Prozessmodell zu integrieren. Alle Probleme, Anregungen, Quick-Fixes etc. werden sofort bei Auftreten von allen Projektbeteiligten dort eingetragen. Abhängig von der Größe des Vorhabens wird die Einführung einer eigenen QM-Rolle notwendig, die sich teamübergreifend mit der Erfassung und Auswertung der in dieser Aktivität erlangten Ergebnisse befasst. Ob eine solche Position notwendig ist, hängt stark von der Projekt- und Organisationsgröße ab (vgl. Abschnitt 6.5). Pflicht ist auf jeden Fall ein Workshop am Ende der Iteration mit Repräsentanten aus den Bereichen des Application- und Domain Engineering, dessen Ergebnisse in einem *Evolutions-Dokument* festgehalten werden.

Ergebnis der Aktivität

Im *Evolutions-Dokument* werden alle Veränderungen am Framework festgehalten, die in der Feedback-Aktivität als notwendig erachtet worden sind. Zusätzlich ist es sinnvoll, regelmäßige Meetings von Architekten und Vertretern des Managements sowie des Marketings abzuhalten, um eventuelle strategische Veränderungen am Framework zu erabeiten und die Ergebnisse mit in das Dokument aufzunehmen. Dazu gehören vor allem die Erweiterung des Frameworks um zusätzliche Domänen, Prozess- und Struktur-Veränderungen usw. Das so erstellte und gepflegte Dokument dient dann als Input der *Domänen-Analyse* oder direkt in der *Framework Implementierung* (siehe dort).

6.3 Rollen und (neue) Aufgaben

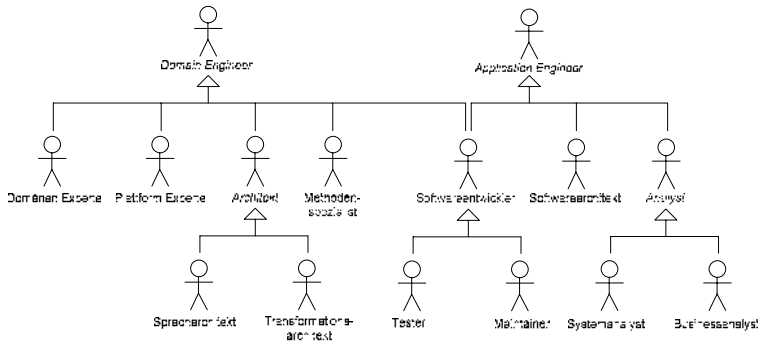
Nachdem in den vergangenen Abschnitten die Kernaufgaben beschrieben wurden, die bei der Durchführung von MDA basierten SEP zu erledigen sind, möchten wir uns im vorliegenden Abschnitt wieder den Projektbeteiligten selbst zuwenden. Was ändert sich an den Aufgaben der „traditionellen“ Rollen, außer dass die Inhaber zum *Engineer* werden?

Die Rollen im Überblick

Abb. 6.4 zeigt die betrachteten Rollen in der Übersicht: Oben die bereits eingeführten abstrakten Rollen *Domain* und *Application Engineer*, die die Basisrollen des betrachteten Prozessmodells

darstellen und die Einteilung der Rollen in die beiden entsprechenden Teilgebiete ermöglichen. Unten einige bekannte und für den einen oder anderen vielleicht auch einige neue Spezialisierungen der beiden Rollen, die wir uns im Folgenden näher anschauen wollen.

Abb. 6.4 Rollen in der MDA



6.3.1 Domain Engineering

Beginnen wir mit den Rollen, die im Bereich Domain Engineering angesiedelt sind und sich, wie in den vorhergegangenen Abschnitten erläutert, mit der Erstellung, Wartung, Pflege und Evolution des MDA-Frameworks beschäftigen.

Domänen Experte (DE)

Der Domänen Experte ist Fachmann auf dem Gebiet einer oder mehrerer Anwendungs- oder Geschäftsgebiete. Er ist im Prozess verantwortlich für die Abstraktion der unterliegenden Konzepte in Domänenmodelle und -beziehungen, die dann selbst als Basis zur Erstellung der korrespondierenden UML-Profile, Metamodelle und Transformationen durch die Architekten dienen (siehe unten).

Wichtige Voraussetzung zur Erfüllung der gestellten Aufgaben ist gelebte Erfahrung in der analysierten Domäne sowie ausreichendes Abstraktionsvermögen zur Bewertung und Einschätzung der Chancen einer Aufbereitung dieses Wissens zur Wiederverwendung innerhalb des Frameworks. Obwohl bei dieser Rolle das Fachwissen die Hauptrolle spielt (und nicht das Design der resultierenden Domänen-Spezifischen Sprachen und Transformationen) ist zur Kontrolle der Ergebnisse nichtsdestotrotz ein Verständnis in Modellierungstechniken und -werkzeugen sowie der verwendeten Modellierungs- und Transformationssprachen Pflicht. Im Falle der Verwendung von UML sind dies beispielsweise die Metamodellierung in MOF, dem Profilkonzept der Infrastructure sowie Sprachen zur

Modellierung von Dynamik und Regeln (ActionSemantics, OCL, …).

Plattform Experte (PE)

Analog zur eben betrachten Rolle des Domänen Experten kümmert sich der Plattform Experte um die Abstraktion von Fachwissen zur Konservierung innerhalb des verwendeten Frameworks. Ihn interessieren jedoch nicht die unterliegenden Geschäftskonzepte oder Anwendungskonstrukte, die diese unterstützen, sondern die Realisierung dieser Anwendungen durch konkrete Technologien (Plattformen). Auch hier gilt, dass die zu erarbeitenden Ergebnisse nicht die Notationen und Transformationen umfassen, sondern vielmehr die Basis für Arbeit der dafür zuständigen Architekten des Domain Engineering bilden, die wir in den folgenden Abschnitten kennen lernen werden.

Wichtigste Aufgabe der PEs ist vor allem die Erstellung einer System(referenz)architektur, deren Kommunikation dem restlichen Entwicklungsteam gegenüber sowie die Aufnahme von Anregungen und Kritik zur Verbesserung und Anpassung dieser Architektur in der weiteren Entwicklung des Frameworks. Zur Erfüllung dieser Aufgabe sind daher auch prominente kommunikative Fähigkeiten und Teamfähigkeit verlangt, die mitentscheidend für den Erfolg des Gesamtvorhabens sind. Da das gesamte Entwicklungsteam mit seinen Ergebnissen arbeiten muss, sind die PEs nach dem Projektleiter die Person, die das Team von der Tragfähigkeit der Referenzarchitektur überzeugen muss. Dazu ist nicht nur die eigene Überzeugung wichtig, sondern auch eine gewisse Fähigkeit verlangt andere von neuen Vorgehensweisen begeistern zu können. Der Plattform Experte muss auf Vorbehalte, die die Realisierbarkeit betreffen, nicht nur technische, sondern auch motivierende Antworten parat haben; eine Kombination, die aus der Erfahrung heraus nicht immer bei Personen mit dem tiefen technischen Wissen, das die Rolle verlangt, gegeben ist und daher die Besetzung zu einer wohlüberlegten Entscheidung machen sollte (vgl. dazu auch die Ausführungen zum *Schweigen im Architektenturm* in Abschnitt 6.6.3).

Spracharchitekt (SpA)

Auf Basis der Analysen der Expertenrollen und der daraus hervorgegangenen Plattform- und Domänenmodelle erstellt der Sprachartchitekt die notwendigen Beschreibungssprachen für die Modellierung innerhalb des SEPs. Dazu gehört sowohl die Erweiterung von UML-Diagrammen durch Profile, als auch die Erstellung von DSLs

durch die Definition komplett neuer Metamodelle mit Mitteln der MOF bzw. vergleichbarer Werkzeuge zur Definition von Sprachen.

Ebenfalls Aufgabe des SpA ist die Erstellung oder Adaption einer Transformationssprache zur Überbrückung der Abstraktionslücken zwischen den Einzeldarstellungen und der daraus resultierenden, automatischen Erzeugung von CIMs, PIMs und PSMs sowie die Auswahl einer Sprache zur Beschreibung der Codegenerierungs-Templates. Neben einem ausgezeichneten Verständnis des Metamodellierungskonzeptes und der Kenntnis mindestens eines Metamodellierungs-Frameworks (z. B. der MOF) gehören auch die Kenntnis der notwendigen Modellierungswerkzeuge in der Tiefe zum Pflichtprogramm des SpA.

Gerade in der Anfangsphase der MDA-basierten Entwicklung muss der SpA also eine der Hauptaufgaben schultern. Bevor es eventuell einmal so weit ist, dass UML-Profile für alle Technologie- und Geschäftsbereiche von der OMG oder ähnlichen Standardisierungsinstituten zur Verfügung gestellt werden, ist die Erstellung derselben bzw. die Suche und Evaluierung von Profilen Dritter seine vornehmliche Aufgabe. Da die von ihm erstellten Sprachen zum Hauptwerkzeug des Application Engineering werden, ist eine enge Abstimmung mit den einzelnen Beteiligten zur Aufnahme der Anforderungen an die jeweiligen DSLs, aber auch zur Pflege der Sprachen in der Evolutions- und Wartungsphase des Frameworks erforderlich. Der SpA muss also, wie die übrigen Architekten- und Expertenrollen des Vorgehens auch, ein hohes Maß an Kommunikationsfähigkeit besitzen. Wenn im Folgenden von *den* Architekten gesprochen wird, sind zur Vereinfachung auch immer die Expertenrollen eingeschlossen.

Transformationsarchitekt (TA)

Aus den Ergebnissen der Analysen der Domänen- und Plattform-Experten und der vom Spracharchitekten zur Verfügung gestellten Transformations- und Templatebeschreibungssprache erstellt der TA die Transformationsbeschreibungen der Modell-zu-Modell (M2M) und Modell-zu-Text (M2T) Transformationen zur Überbrückung der Abstraktionslücken zwischen den einzelnen Darstellungen. Neben der Erstellung der Beschreibungen in Zusammenarbeit mit den jeweiligen Expertenrollen sind auch die Spezifikation und Realisierung der entsprechenden Software wie Transformatoren, Reverse-Engineering Werkzeuge usw. Aufgabe des TA.

Sein Profil verlangt daher nicht nur grundlegende Kenntnisse der Quell- und Zieldomänen der Transformation, sondern auch die

theoretischen Grundlagen von Transformationen und deren praktische Umsetzung in lauffähige Software.

Aufgrund der hohen Belastungen dieser und der restlichen Architektenrollen und des hohen Abstimmungsbedarfs mit großen Teilen des restlichen Domain Engineering Teams gerade in den Anfängen der Entwicklung mit MDA ist unter Umständen die Bildung eines Architektur-/Kern-Teams sinnvoll, das die Aufgaben im Vorfeld des Projektes unter sich aufteilen kann. Auch hier dient die Granularität der vorgeschlagenen Rollen lediglich als Empfehlung und muss auf die individuellen Bedürfnisse angepasst werden. So kann, gerade in Referenz- und Pilotprojekten, die Aufteilung einer oder mehrerer Architektenrollen in die Rollen Technischer Projektmanager, Architekt und Leitender Entwickler sinnvoll sein. Bei der Entscheidung, wie dies im Einzelfall aussehen kann, hilft die Etablierung einer dezidierten Rolle zu diesem Zweck, die wir im nächsten Abschnitt vorstellen.

Methoden Spezialist (MeS)

Der Methoden Spezialist kümmert sich um die Anpassung bereits im Unternehmen bestehender SEP an die neuen Erfordernisse der MDA und legt die notwendigen Rollen, Aktivitäten und Artefakte fest, die in den beiden Engineering-Bereichen eine Rolle spielen. In der Anwendungsentwicklung (Application Engineering) ist der MeS vor allem Berater des verantwortlichen Projektmanagers und hilft ihm beim Tailoring und der Durchführung des Projektes nach den Richtlinien des gewählten Vorgehens- und Prozessmodells.

Neben der ständigen Ansprechbarkeit als Mediator bei Problemen innerhalb der einzelnen Projekt- und Iterationsphasen sollte der MeS auch bei den *Lessons-Learned* Phasen am Ende von Iterationen dabei sein, um eventuelle Anpassungen am Gesamtprozessmodell vornehmen zu können (vgl. Abschnitt 6.2.6), und so die kontinuierliche Optimierung des Modells zu gewährleisten.

Zusätzliche Aufgabe des MeS kann es sein, eventuelle initiale Ängste und Vorbehalte von Projektbeteiligten zu entkräften. Weit verbreitet ist beispielsweise die Vermutung, MDA sei nur dazu konzipiert, um möglichst viele Arbeitsplätze innerhalb der Softwareentwicklung einsparen zu können. Zusätzlich wird gerade von den beteiligten Domänen Experten (z. B. Spezialisten aus den Fachbereichen eines Unternehmens) oft Skepsis darüber geäußert, ob es überhaupt möglich sein kann das neu benötigte Wissen um Modellierungskonzepte und ungewohnte Notationen in vertretbarer Zeit und nicht ausuferndem Budget zu erlangen. Diese und ähnliche Vorbehalte entstehen meist durch Unkenntnis dessen, was auf die

Beteiligten zukommt. Hier kann durch Vorgespräche, Schulungen und ständigem Coaching während des Projektes, die den Betroffenen die neue Arbeitsweise vermitteln, Klarheit geschaffen und damit oft auch die Vorurteile beseitigt werden.

Aktivität: Ins-Boot-holen

Die Notwendigkeit, alle Beteiligten „ins Boot zu holen", sollte nicht unterschätzt werden. Das Gelingen des Vorhabens MDA mit der angestrebten Modellkette vom Geschäftsprozess bis zur fertigen Anwendung kann nur gelingen, wenn alle Beteiligten an einem Strang ziehen. Auch wenn nicht alle Projektbeteiligten von Anfang an vom Erfolg des Vorhabens überzeugt werden können, muss zumindest die Bereitschaft eingeholt werden es zu versuchen. Diesen Status-quo zu erlangen, ist allerdings gerade in großen und etablierten Unternehmen nicht einfach und erfordert, neben den schon so oft erwähnten kommunikativen Fähigkeiten, ebenfalls einen gehörigen Schuss diplomatisches Geschick und die notwendige Zeit, auf die Wünsche und Fragen der Beteiligten eingehen zu können. Obwohl die Geldgeber eines Projektes oft von der Notwendigkeit einer selbstständigen Rolle und deren Besetzung durch eine Vollzeitkraft erst überzeugt werden müssen, hat sich diese Maßnahme in vielen Projekten nicht nur bewährt, sondern oftmals erst deren erfolgreiche Durchführung ermöglicht.

6.3.2 Application Engineering

Die meisten Rollen dieses anwendungsentwickelnden Bereichs behalten in der MDA alle Aufgaben, die auch „traditionell" zu ihrer Beschreibung gehören. Was sich ändert, ist weniger die Art der Aufgaben – die Analyse, Entwicklung sowie das Testen und die Pflege von Anwendungen – als vielmehr der Ablauf und die Entwicklungsumgebung, die zu ihrer Erfüllung notwendig sind.

War es nach der Analyse bislang eher so, dass der Architekt den Bauplan vorgab und die Entwickler von Null auf mit hohen Freiheitsgraden die Realisierung dieser Pläne in handfesten Code zu verfolgen hatten, werden innerhalb der MDA die Werkzeuge und eine Referenz vorgegeben, nach der sich die Entwicklung richten muss.

Durch die Anwendung einer Referenzarchitektur und die automatische Transformation der Analysemodelle in einen Designrahmen, der dieser Referenz folgt, wird die Aktivität des Designens der Anwendung von einer Folge vieler kleiner Entscheidungen mehrerer Einzelpersonen (Architekten und Entwickler) in großen Teilen von der Aktivität der Transformation abgelöst. Was dies für die einzelnen Rollen bedeutet und welche zusätzlichen Aspekte eine Rolle spielen wollen wir im Folgenden betrachten.

Software- und Business Analyst (SAn, BAn)

Die vornehmlichen Aufgaben der Analysten: die Aufnahme der funktionalen und nichtfunktionalen Anforderungen des Systems, bleiben erhalten und ändern sich in der Art der Durchführung im Prinzip nicht. Was sich ändert, ist jedoch, dass die Ergebnisse der Analyse in festgelegter Modellform aufgeschrieben werden. Diese Modelle bekommen neben dem Kommunikationscharakter, der im Kapitel Modellierung unter dem Stichwort *Sketch-Modelle* aufgetaucht ist, nun auch formalen Charakter und werden direkt zur Generierung von Quellcode bzw. der Transformation in weniger abstrakte (Modell-)Ebenen verwendet.

Formalisierung im Umgang mit Modellen

Diese Formalisierung im Umgang mit Modellen stellt eine Kernänderung der Aufgaben dar, die an die Analysten gestellt werden. Die Bereitschaft zur formaleren Arbeit durch den Analysten selbst sowie die Bewilligung von Zeit und Budget zur Schulung im sicheren Umgang mit diesem Formalismus durch das Management, sind unbedingtes Muss, da die Stabilität des gesamten Entwicklungsprozesses mit der Korrektheit der Modelle steht und fällt. Wie bereits im Abschnitt über die Rolle des Methoden Spezialisten angeklungen ist, ist diese Bereitschaft oftmals nicht a priori vorhanden, sondern erfordert Überzeugungsarbeit und diplomatisches Geschick.

Software Architekt (SoA)

Auch im MDA-SEP bleiben die Entscheidungen über die benutzten Technologien und verwendeten Architekturpatterns in der Hand des Architekten. Anders als in herkömmlichen Vorgehensweisen hat er jedoch nicht mehr völlig freie Hand, sondern muss sich einerseits an die vom Framework vorgegebenen Zielplattformen halten (bzw. die Aufnahme weiterer Plattformen anregen) und beeinflusst das Design nun nicht mehr ausschließlich direkt, sondern zum großen Teil durch die geeignete Auswahl von erstellten Modellen, auf die dann Transformationsvorschriften angewendet werden. Die Auswahl dieser Transformationen sowie die Durchführung der notwendigen projektspezifischen Parametrisierungen gehören ebenfalls zum Aufgabenpaket des Software Architekten.

Müssen Artefakte des Frameworks, wie bspw. die eben angeführten Transformationsvorschriften, angepasst oder verbessert werden, ist die Kommunikation dieser Change-Requests an die zuständigen Rollen des Domain Engineering auch eine der Verantwortlichkeiten, die im Rahmen dieser Rolle erfüllt werden wollen.

Die Ausübung der Aufgaben des SoA ändert sich also wesentlich. Der Designprozess findet durch die Auswahl und Parametrisierung von Transformationen aus dem Fundus des Frameworks sowie der passenden Modelle aus der Arbeit der Analysten und Entwickler und

nicht mehr ausschließlich als geistige Einzelleistung des jeweiligen Architekten/Entwicklers statt. Dadurch wird die Referenzarchitektur forciert, Spezialistenwissen multipliziert und eine Vereinheitlichung des Designs ermöglicht. Weiterhin braucht der Architekt sich nicht immer wieder mit der Lösung derselben Probleme herumzuschlagen und kann sich auf die Bearbeitung der relevanten Geschäftsanforderungen konzentrieren. Einmal gelöst, wird die Lösung von „Standardproblemen“ als Muster (Transformationsvorschrift) von den Plattformexperten in das MDA-Framework aufgenommen und steht dann allen Architekten des Unternehmens zur Verfügung.

Software Entwickler (SE)

Ausgangspunkt für die Arbeit des Software Entwicklers sind die plattform-spezifischen Modelle (PSMs), die durch die Anwendung der vom Architekten ausgewählten Transformationen auf die ebenfalls zur Auswahl gehörigen Modelle entstehen. Im Falle der M2T Transformationen sind dies Textartefakte wie Deployment-Deskriptoren oder Quellcoderahmen, in denen der Entwickler die nicht generierten Teile der Applikation vervollständigen muss.

Diese Vervollständigung kann durch die Einbettung des nicht generierten Codes in Protected-/User-Sections geschehen oder durch die Generierung eines domänen-spezifischen Frameworks, in dem der Code dann durch Vererbung unter Zuhilfenahme polymorpher Techniken eingebunden wird (vergleiche dazu die Ausführungen im Kapitel 5 – Transformation).

No Monkey-Work please!

Obwohl die gerechtfertigte Befürchtung besteht, muss die Arbeit des Entwicklers jedoch keinesfalls zur *Monkey Work* (vgl. hierzu die Ausführungen zu *Code Monkeys/Monkey Work* in Abschnitt 6.6.3) degenerieren. Da der Coderahmen immer aktuell aus bereits bestehenden Modellen generiert wird, gehört auch die Pflege von (Teil-)Modellen zu den neuen Entwickleraufgaben. Änderungen an Modellen, die die Anwendung von Design-Patterns o. Ä. betreffen sind jetzt direkte Aufgaben des Entwicklers. So wird die Vorstellung der „graphischen Programmierung“ wieder ein Stück mehr Realität. Wie eine solche Modellierung von (Teil-)Algorithmen aussehen kann, werden wir später noch am praktischen Beispiel sehen.

Auch das Refactoring geeigneter Modelle sowie die Erstellung und das Tuning der Transformationen und Anregungen zur Evolution des Gesamtframeworks, zusammen mit dem Software Architekten, sind wichtige Aufgabenbereiche des Entwicklers. Gerade das Refactoring im Großen erhält durch die visuell-formale Darstellung der Systemarchitektur neue Möglichkeiten. Der Entwickler wird so zum Architekten seines eigenen Zuständigkeitsbereiches. Dieser

Zuständigkeitsbereich kann auf Paket-/Komponenten- oder auf Subsystemebene angesiedelt sein. Während der Architekt das Grobkonzept vorgibt, füllt der Entwickler dieses mit Leben, indem er die Vorgaben durch Anwendung geeigneter Entwurfsmuster und der Überbrückung der Abstraktionslücken der Modelle hin zur lauffähigen Anwendung erfüllt.

Ob Diagramme oder Quellcode, wie wir gesehen haben, sind sie letztendlich alle Modelle des zu erstellenden Systems. Programmieren wird zu Modellieren und umgekehrt. Die ursprünglich strikte Trennung, die das Modellieren von Systemen fast ausschließlich Architekten, Designern und Analysten vorbehielt, verwischt. Anforderungen, die durch den zunehmenden Formalisierungsgrad der Modelle entstehen, und die schon beim Architekten zu Schulungs- und Budgetbedarf führten, treffen für die Rolle des Entwicklers im selben Maße zu.

Das Feedback des Entwicklers gegenüber dem Domain Engineering Team ist wesentlich für die erfolgreiche Durchführung der MDA. Ideen, die auf hoher abstrakter Ebene, quasi auf dem Reißbrett, entstanden sind, bewähren sich gerade unter den Händen der Entwickler in der „harten“ technischen Realität. Nur durch enge Zusammenarbeit aller Beteiligten können Unzulänglichkeiten in den Transformationen, den Beschreibungssprachen bzw. in den Templates oder den unterliegenden Prozessen, sprich: in der MDA-Infrastruktur, erkannt und effizient optimiert werden.

Wie bereits bei anderen projektbeteiligten Rollen erwähnt gehört nicht zuletzt auch die Bereitschaft des Entwicklers, diese Aufgaben auch wahrnehmen zu wollen bzw. sich die benötigten Kenntnisse anzueignen zur Grundvoraussetzung einer erfolgreichen MDA-Einführung. Entwickler, die nicht bereit sind ihre Tätigkeit auf ein höheres abstrakteres Niveau zu verlagern, wie es der MDA-Ansatz verlangt, sind für die kommenden Aufgaben der Softwareentwicklung und der IT im Ganzen nur bedingt einsatzfähig. Diese Gruppe macht nach unserer Überzeugung jedoch nur eine Minderheit aus. Neue Aufgaben beinhalten auch neue Herausforderungen und machen die Entwicklungstätigkeit nicht langweiliger, sondern eher interessanter.

Tester (Te)

Tester bekommen in der MDA die Möglichkeit, Testskripte direkt aus den Modellen zu generieren, die dann direkt gegen die generierten oder manuell erstellten Teile der Applikation ausgeführt werden

können. So kann die mühsame Arbeit der Erzeugung von Testkonzepten und –daten durch Automatisierung erleichtert werden.

Ein Notationsmittel, das die direkte Modellierung von Tests in UML erlaubt, ist das UML2-Testing-Profile [U2TP]. Eclipse bietet mit dem *Eclipse Test & Performance Tools Platform Project* [TPTP] bereits erste Unterstützung dieses Profils und wird von einer Reihe großer Firmen mit Rat und Tat gesponsert, sodass dort in naher Zukunft große Fortschritte zu erwarten sind.

Maintainer (Ma)

Die Wartung von Anwendungen macht Schätzungen zufolge mehr als die Hälfte der Kosten aller Softwaresysteme aus ([SPL03] spricht sogar von 90 Prozent). Die Aufgabe der Maintainer wird vor allem dadurch erschwert, dass nahezu unmittelbar nach der Fertigstellung des Systems das Wissen um die Integration des Systems in den Gesamtkontext des Unternehmens schwindet und durch Änderungen anderer abhängiger Systeme schließlich zum hoffnungslosen Unterfangen wird.

Die Auswirkungen von Änderungen am System auf andere Anwendungen und die abhängigen Geschäftsprozesse können dann nur noch schwer oder gar nicht abgeschätzt werden. Auch das Wissen um die interne Struktur des Systems schwindet schnell, sodass notwendig werdende Anpassungen der Anwendung an neue Geschäftsanforderungen immer schwieriger werden.

Werden in der MDA plattform-spezifischere Modelle aus weniger abstrakten Modellen generiert, lassen sich die Zusammenhänge der Einzelkomponenten sowie die Abhängigkeiten zu anderen Systemen jederzeit aus den entsprechenden Teilmodellen ablesen. Werden zudem die Geschäftsprozesse modelliert (beispielsweise in Form eines CIMs) können auch die Abhängigkeiten zu Geschäftsprozessen modellhaft erfasst werden. Dies garantiert eine längere Lebenszeit des Systems, da Anpassungen und selbst schwerwiegende Änderungen wie der Wechsel unterliegender Technologien, Hardware o. Ä. jederzeit durch eine Anpassung der Modelle bzw. der Änderungen der verwendeten Transformationen, möglich werden. Die initialen Investitionen bei Einführung der MDA stellen somit ein Investment in die Lebensdauer und die Wartbarkeit dar, das sich auch beim Eintritt in die Wartungsphase der Applikation bezahlt macht.

6.3.3 Fazit

Durch die Einführung eines MDA-Ansatzes in einen bestehenden Software Entwicklungsprozess bleiben viele Aufgaben des alltäglichen Projektgeschäftes unverändert. Zusätzlich kommen jedoch einige Aktivitäten hinzu, die neue Herausforderungen darstellen und die Erarbeitung neuer Kenntnisse von allen Beteiligten sowie die notwendige Unterstützung und Rückhalt vom übergeordneten Management verlangt (vgl. *Entscheidungen ohne Konsequenz* in Abschnitt 6.6.3).

Auf unser Vorhaben bezogen bedeutet dies, dass nur ein motiviertes Team einen MDA-getriebenen-Ansatz zum Erfolg führen kann. Eventuell nötige Überzeugungs- und Schulungsarbeit ist unbedingt im Vorfeld zu leisten. Fehlt diese Arbeit, oder wird sie nur halbherzig durchgeführt, sind erste Gehversuche im Umfeld MDA mit hoher Wahrscheinlichkeit schon im Ansatz zum Scheitern verurteilt. Dies ist keine neue Erkenntnis. [Coc01] und [DL99] widmen sich diesem Thema ausführlich und geben interessante Tipps zur Motivation und Heranführung von Mitarbeitern an neue, ungewohnte Aufgaben.

Nur durch vorhandene Motivation und die enge Zusammenarbeit aller Beteiligten kann der Versuch gelingen, einen MDA-Ansatz möglichst pragmatisch umzusetzen. Die Hervorhebung dieses kommunikativen Aspektes, der in allen bisher gegebenen Tätigkeitsbeschreibungen angeklungen ist, macht erneut die Bedeutung des agilen Charakters in der Sichtweise auf MDA deutlich, die durch dieses Buch bekräftigt werden soll. Im *Agilen Manifest* [AM] heißt es:

> *Individuen und Interaktionen sind wichtiger als Prozesse und Werkzeuge.*

Wie wir gesehen haben, wird sich dieser Aspekt auch im vorgeschlagenen Vorgehen zum Glück nicht ändern.

6.4 Einführung von MDA ins Unternehmen

Nachdem wir im ersten Teil des Kapitels quasi das Dach eines bereits laufenden MDA-Prozesses gelüftet und einen kurzen Blick in die rollenbezogenen, dynamischen Charakteristiken gewagt haben, werden nun organisatorische Aspekte beleuchtet, die bei der Einführung von MDA in eine bestehende Unternehmensstruktur eine Rolle spielen. Auch hier wird versucht, zur Lösung der anstehenden

Probleme die geeigneten Kernprinzipien zu identifizieren bzw. geeignete weiterführende Literatur als Hilfestellung zu benennen.

MDA-Einführung als Technologiewechsel

Die Einführung eines MDA-basierten Ansatzes in ein Unternehmen, in dem bereits Strukturen und Prozesse zur Entwicklung von Software etabliert sind, ist ein Spezialfall eines *Technologie Change Projects (TCP)*. Angelehnt an die Betrachtung der Besonderheiten eines solchen Projektes in [CN01] wollen wir uns im Folgenden anschauen, welche Analogien zur Einführung der Model-Driven Architecture gezogen werden können:

Kernaufgaben bei Technologiewechseln

Der Charakter einer solchen Migration hin zum neuen Vorgehen, und damit auch die Auswahl der durchzuführenden Schritte und Maßnahmen, die eine solche zum Erfolg führen, ist hochgradig von der individuellen Situation abhängig und macht so die Angabe eines detaillierten Kochrezeptes zur Durchführung notwendigerweise ungeeignet. Im Kern sind die folgenden Schritte jedoch allen Projekten dieser Art gemein:

- Die *Positionsbestimmung* des Unternehmens: Wo stehe ich in Bezug auf die einzusetzende Technologie im Augenblick?
- Die Definition der zu erreichenden *Ziele (Vision)*: Wo möchte ich am Ende der Einführungsphase, bzw. am (geplanten) Ende meiner Bemühung stehen?
- Der Entwicklung und Durchführung einer *Strategie*, um die Kluft zwischen *Ist* und *Soll* zu überbrücken.

Probleme in der Einführungsphase

Doch selbst, wenn eine erste Vision im Raume steht, die Verantwortlichen und die Mitarbeiter überzeugt sind und erste finanzielle Zusagen des Managements zur Durchführung des Projekts zur Verfügung stehen, gibt es noch genug Steine die auf dem Weg zur endgültigen Umsetzung aus dem Weg geräumt werden wollen:

Jetzt gehts los?

In florierenden Unternehmen sind die Angestellten nahezu alle damit beschäftigt aktuelle Projekte abzuwickeln. Projekte, die mit harten Deadlines zu kämpfen haben, sodass für eine Einführung eines neuen Ansatzes nahezu keine qualifizierten Kapazitäten zur Verfügung stehen, die sich um die Umsetzung der definierten Ziele kümmern können. Dies gilt insbesondere für die hochqualifizierten Spezialisten und führenden Entwickler, bei denen eine fast 100%ige Auslastung zu erwarten ist. Gerade diese Mitarbeiter sind es jedoch,

die zur wesentlichen Minimierung des Risikos bei Projekten mit hohem Risikopotenzial, zu denen die TCPs mit Sicherheit gehören, beitragen können. Hier gilt es die benötigten „Spielräume“ zu schaffen, um den Erfolg des Vorhabens nicht zu gefährden (Tom DeMarco spricht in [DeM01] von der Notwendigkeit des Vorhandenseins von *Slack*). Notorische Ressourcenknappheit ist also eines der Haupthindernisse, die zu Anfang eines TCPs aus dem Weg geräumt werden müssen.

6.4.1 Ad-hoc-Vorgehen oder Iterative Einführung?

Will man sich nicht von seinem Plan abbringen lassen, bleiben nur zwei Möglichkeiten mit einer solchen Situation umzugehen:

- der Umsetzung der Ziele *Ad hoc*,
- oder der *schrittweisen Einführung* der neuen Technologie, die wir im Folgenden *Iterative Einführung* nennen wollen.

Schauen wir uns die beiden Alternativen einmal näher an:

Ad-hoc-Vorgehen

Keine Kompromisse!?

Wie der Name schon vermuten lässt, handelt es sich bei dieser Strategie um die aggressivere Variante eines TCPs: Alle aktuell laufenden Projekte werden auf Eis gelegt, bis der Wechsel stattgefunden hat oder sogar beendet ist, um die notwendigen Kapazitäten und Freiräume zur Migration zu schaffen. Die gesamte Belegschaft konzentriert sich also während der Einführungsphase ausschließlich auf die Umsetzung der gesteckten Ziele.

Vorteile: Diese Strategie ist der kürzeste Weg zum Ziel und damit auch der Weg mit der kürzesten Zeitspanne bis zum Genuss der versprochenen Vorteile und des ROI. Zudem sind alle Mitarbeiter gleichzeitig und von Anfang an Teil der neuen Prozesse und Arbeitsweisen und bekommen so die Möglichkeit die Erfahrungen und Einsichten, die während der Umstellung gewonnen werden, aus erster Hand mitzuerleben. Die Identifikation mit den auf diese Weise quasi selbst entwickelten Techniken bewirkt eine starke persönliche Verpflichtung dem Erreichen der Ziele gegenüber und garantiert so Interesse an der konsequenten Einhaltung und Verbesserung des neuen Ansatzes.

Weiterer netter „Nebeneffekt“ dieser mutigen Vorgehensweise ist, dass auch die Notwendigkeit der nachträglichen „Synchronisa-

tion“ von Einheiten, die bereits mit dem neuen Ansatz arbeiten, mit solchen, die während der Einführung noch die „herkömmliche“ Arbeitsweise beibehalten haben entfällt: So werden weitere unnötige Reibungsverluste verhindert.

Kosten und Risiken: Wie zu erwarten, kommen die Vorteile, die ein solch radikaler Schritt mit sich bringt, nicht ohne Preis: Zusätzlich zu den in jedem Fall zu erwartenden Umstellungskosten für Personal und Infrastruktur gesellen sich die fehlenden Einnahmen durch beendete oder abgelehnte Projekte. Auch eventuelle Verluste in der Kundenzufriedenheit können sich negativ durch nicht erteilte Folgeprojekte bemerkbar machen. Im schlimmsten Falle scheitert die Migration, ein Risiko, das bei Eintritt nahezu unausweichlich schweren Schaden am Unternehmen hinterlässt.

Kompensationsmöglichkeiten: Stehen zum Zeitpunkt der geplanten Umstellung nur wenige Projekte an und bewegen sich die Fertigstellungszeiträume in adäquaten Bahnen, können diese eventuell zeitnah, bereits unter Nutzung der neuen Entwicklungsprozesse, fertig gestellt werden. Ist dies nicht möglich, können (Teil-) Projekte outgesourced werden, um Freiräume zu schaffen. In manchen Fällen stehen Forschungsbudgets zur Verfügung, deren verstärkte Inanspruchnahme zur Überbrückung der „mageren Umstellungszeit“ dem Management eventuell schmackhaft gemacht werden kann.

Zur Entscheidungsfindung, ob man einen solch radikalen Schritt denn wirklich gehen will, ist die Durchführung eines *Pilotprojektes* mit folgender Analyse der dort gewonnenen Erkenntnisse als Entscheidungsgrundlage empfehlenswert. Dieser besonderen Art von Projekten widmen wir uns in Abschnitt 6.4.2.

Iterative Einführung

Die weniger radikale Lösung

Ist die oben beschriebene Strategie im eigenen Unternehmen nicht durchführbar oder wird eine Lösung mit weniger Risiko angestrebt, besteht die Möglichkeit einer schrittweisen Einführung des neuen Ansatzes. Dazu wird zuerst ein kleines Kernteam gebildet, das mit folgenden Aufgaben betreut wird:

- der Analyse der Anforderungen,
- der Entwicklung einer geeigneten Infrastruktur sowie
- der Entwicklung geeigneter Prozess- und Organisationsanpassungen.

Das Kernteam beginnt sofort mit der Abarbeitung dieser Aufgaben, während die übrigen Mitarbeiter ungestört ihrem Projektalltag nachgehen können.

Vorteile: Wird diese Strategie gewählt, liegt der größte Vorteil darin, dass das laufende Projektgeschäft von der Umstellung nicht bzw. nur minimal beeinflusst wird. Wird die Möglichkeit genutzt, zusätzlich vom Kernteam fertig gestellte Teilergebnisse einzeln in diesen laufenden Projekten auszuprobieren, erhalten diese ihr eigenes Pilotprojekt (siehe auch Abschnitt 6.4.2), was die Qualität der entwickelten Infrastruktur weiter verbessert. Zur Erprobung in Frage kommen hier beispielsweise Profile, Modellierungswerkzeuge, Prozessverbesserungen, etc. Insgesamt findet die Umstellung also nicht mit dem berüchtigten „Big-Bang" statt, sondern alle Mitarbeiter und Prozesse haben Zeit sich den neuen Anforderungen anzupassen und erste Fehler unter weniger risikoreichen Bedingungen zu finden und zu beheben.

Kleiner Aufwand, k(l)eine Resultate?!

Risiken: Frei nach dem Sinnspruch „Kleiner Aufwand, kleine Resultate!" besteht die Gefahr, dass das Kernteam der Fülle von Aufgaben nicht gewachsen ist und damit die Umstellungsziele an sich in Gefahr geraten. Zudem werden solche Kernteams manchmal als etwas wahrgenommen, was dem eigentlichen Unternehmenszweck – dem bereits angesprochenen „Geldverdienen" – nebenher läuft, und werden so oft nicht mit dem notwendigen Ernst betrachtet bzw. in ausreichendem Maße in ihren Aufgaben unterstützt. Die Möglichkeit, den neuen Ansatz als eine Chance für die Verbesserung der Zukunftsfähigkeit des Unternehmens zu kommunizieren, wird so vertan. Skeptiker, die von vorneherein der Durchführung ablehnend entgegenstanden, erhalten Zeit, Strategien zu entwickeln um die neuen Maßnahmen zu unterlaufen und damit das Gesamtziel in Gefahr zu bringen.

Fallen die Ergebnisse des Kernteams trotz dieser ungünstigen Vorbedingungen positiv aus, ist wegen ihrer geringen Größe im Vergleich mit den Durchschnittsprojekten des Unternehmens immer noch nichts darüber ausgesagt, ob die erarbeiteten Prozesse und Methoden auch in größerem Rahmen skalieren. Hier ist man also trotz durchgeführtem „Ausprobierens" nicht vor negativen Überraschungen sicher (vgl. hierzu die Ausführungen zu der idealen Projektgröße und –zusammenstellung von Pilotprojekten in Abschnitt 6.4.2).

Kosten: Auch hier fallen die angesprochenen initialen Umstellungskosten an, die notwendigerweise mit einer solchen Unternehmung

verbunden sind. Zusätzlich verzögert sich der Zeitpunkt, an dem die (finanziellen) Früchte des neuen Ansatzes geerntet werden können, was implizite Zusatzkosten bedeutet.

Kompensation: Die angesprochenen Risiken können nur kompensiert werden, indem das Kernteam seinen Aufgaben gemäß zusammengesetzt und budgetiert wird. Nur die kompetentesten Angestellten des Unternehmens kommen für eine solche Aufgabe in Frage, was im Normalfall die Freistellung der Mitarbeiter aus aktuell laufenden Projekten bedeutet und harschen Protest der verantwortlichen Projektleiter zur Folge haben wird. Neben der Bereitstellung des notwendigen Budgets ist außerdem ein unbedingter Rückhalt in der Führung des Unternehmens erforderlich sowie die klare Kommunikation der unweigerlich kommenden Verbindlichkeit der entwickelten Techniken und Prozesse. Um diese und die zuvor angegebenen Risiken zu minimieren, empfiehlt es sich, der Anfangsphase der iterativen Einführung ein oder mehrere Pilotprojekte zur Seite zu stellen, deren Charakteristiken wir im Folgenden vorstellen wollen.

6.4.2 Pilotprojekte

Zusammen mit der iterativen Einführung, aber auch als qualifizierter „Testballon“ für Ad-hoc-Umstellungen, können so genannte Pilotprojekte eingesetzt werden, um die inhärenten Risiken eines Technologiewechsels einzudämmen. Nach einer ersten Initialphase des Kernteams mit der neuen Technologie müssen sich die geschaffenen Werkzeuge und Prozesse im Rahmen eines Realprojektes beweisen und werden notfalls noch während des Projektes verfeinert bzw. verbessert.

Die Vorteile einer solchen Vorgehensweise ergeben sich vor allem in zwei Punkten [Ver02]:

- **Risikominimierung**: Durch das Sammeln erster Erfahrungen durch ein Pilotprojekt unter realen Bedingungen wird ein *Proof-of-Concept* [POC] geschaffen, das sowohl die Umsetzbarkeit, als auch die Sinnhaftigkeit des gesamten Unterfanges vor der Einführung im gesamten Unternehmen unter Beweis stellt. Schlägt das Projekt fehl, können auch gravierende Korrekturen ohne größeren Schaden vorgenommen bzw. vielleicht sogar die Entscheidung zur vorzeitigen Beendigung des Unterfangens getroffen werden.

- **Optimierungspotenzial**: Mögliche Verbesserungen vor der generellen Einführung können am besten durch reale Erfahrungen angeregt werden. Obwohl die ständige Verbesserung der Methoden und Werkzeuge sowieso Ziel eines jeden Unternehmens sein sollte, gelingt dies innerhalb eines Pilotprojektes unter wesentlich risikoärmeren Bedingungen. Auch Ideen, von deren Umsetzbarkeit nicht alle Verantwortlichen sofort überzeugt sind, können hier ausprobiert werden.

Erfolgsfaktoren: Während das Kernteam, wie oben erwähnt, nur aus ausgewählten Spezialisten bestehen sollte, ist für ein geeignetes Pilotprojekt genau die Mischung der Projektmitglieder entscheidender Erfolgsfaktor. Das Team sollte aus einem, dem Unternehmen typischen, durchschnittlich qualifizierten Personal bestehen, was bedeutet, dass auch weniger erfahrene Mitarbeiter eingesetzt werden müssen. Dies entspricht in vielen Fällen dem täglichen Projektalltag und verdeutlicht auch Haken und Ösen des neuen Systems, die von zu alten Hasen vielleicht unentdeckt „wegkompensiert" würden.

Auch die Projektgröße ist entscheidend für die Aussagekraft der Pilotprojektergebnisse und sollte im Durchschnitt der normalerweise im Unternehmen durchgeführten Projekte liegen. Nicht zu klein, um bedeutsame Ergebnisse liefern zu können, die zur finalen Entscheidung „MDA, Stop oder Go" herangezogen werden können, aber auch nicht zu groß, um dem Risiko eines – eingeplanten – Scheiterns nicht übermäßige Folgen nacheilen zu lassen. Eine Teamgröße von ca. 5-10 Beteiligten hat sich hier bewährt. Die Größe sollte auf jeden Fall ausreichen, um das Rollenkonzept des angebundenen Prozesses umsetzen zu können.

6.4.3 Fazit

Um eine schnelle Durchführung des Wechsels zu gewährleisten, und alle Mitarbeiter in einem Boot an die Ufer des neuen Ansatzes zu bringen und von der Wirksamkeit zu überzeugen eignet sich am besten der aggressive Wechsel nach dem „*Alles-oder-nichts*"-Prinzip. Unternehmen, die eine große Anzahl anstehender Projekte mit engen Zeitfenstern abzuarbeiten haben und nicht das Budget oder die notwendige Marktposition besitzen diesen Weg einzuschlagen und das mit diesem Vorgehen verbundene hohe Risiko einzugehen, sind mit einer inkrementellen Einführung besser bedient. Pilotprojekte können bei der Entscheidungsfindung helfen oder innerhalb einer iterativen Einführung die initiale Speerspitze bilden.

Welchen Weg man auch wählt, entscheidend ist die konsequente Durchführung der einmal getroffenen Entscheidung sowie ein

garantierter Rückhalt der Maßnahmen im verantwortlichen Management. Strategische und vor allem bindende Entscheidungen sind zu treffen und zu verfolgen. Halbherzige Versuche, ob in kleinem oder großem Rahmen, sind mit nahezu 100%iger Wahrscheinlichkeit dem Scheitern verurteilt und sollten allen Beteiligten besser von vorneherein erspart bleiben.

6.5 Anpassen bestehender Organisationsstrukturen

Ein Aspekt, der bei der Einführung einer MDA-Strategie in ein Unternehmen kritische Bedeutung besitzt, jedoch in der Literatur oft wenig bis gar keine Beachtung findet, ist der der eventuell notwendig werdenden Anpassungen der Unternehmensstruktur. Da sowohl die Bewertung der Notwendigkeit von Veränderungen, als auch die konkrete Beschreibung derselben stark von der individuellen Situation des betrachteten Unternehmens abhängt, werden an dieser Stelle keine konkreten Handlungsanweisungen gegeben, sondern vielmehr generelle Betrachtungen des Themas angestellt. Diese können dann in der konkreten Situation als Entscheidungshilfe bzw. Anknüpfpunkt für weitere Recherchen dienen.

Die Betrachtungen sind dabei stark den Ausführungen von Clements und Northrop aus [CN01] sowie den Betrachtungen von Bosch zu Organisationsstrukturen bei der Einführung von Software-Produktlinien entlehnt [Bos01]. Auch im gleichnamigen Buch [BKPS04] befasst sich ein eigenes Kapitel ausführlich mit diesem Thema und bietet die Möglichkeit zur Vertiefung der hier angerissenen Aspekte.

Ausgangssituation

Die Organisationsstruktur vieler Unternehmen, die keine explizite Wiederverwendungsstrategie und damit auch keine auf MDA basierenden Entwicklungsansätze verfolgen, sieht so aus, dass für jedes Projekt ein Individualteam für die Dauer des Entwicklungszyklus gebildet wird, das dann selbstständig, unter Verantwortung des jeweiligen Projektmanagers, die volle Verantwortlichkeit für alle das Projekt betreffenden technischen Entscheidungen trägt. In MDA-Kontexten existiert jedoch eine Dualität gleichzeitiger Weiterentwicklung des Frameworks sowie der unter Nutzung von Elementen dieser Infrastruktur parallel entwickelten Kundenprojekte. Dies macht eine Organisationsstruktur notwendig, die nicht mehr projekt-/produktzentrisch angelegt sein darf, sondern auch den Überblick über die Verwendung und das Management eventuell notwendiger Evolution der Assets des Frameworks ermöglichen muss.

Die Evolution umfasst vor allem die Anpassung von Elementen der Infrastruktur durch neue Anforderungen der Individualprojekte, aber auch die Integration neuer Technologien zur Sicherung von strategischen Wettbewerbsvorteilen. Anzupassende Assets können Werkzeuge wie Generator/Transformator, Editoren, aber auch Softwarekomponenten im herkömmlichen Sinne sowie Testpläne, Prozessmodelle usw. sein. Veränderungen an diesen Komponenten sind nur dann gestattet, wenn sie mit der zuständigen Organisationseinheit abgesprochen und von dieser als notwendig befunden worden sind. Notwendig gewordene Änderungen werden dann generalisiert in die Infrastruktur übernommen und das gewonnene Wissen auch den Mitgliedern der restlichen Projektteams zugänglich gemacht. Geschieht eine solche Überwachung nicht, geht nicht nur die Möglichkeit wertvollen Know-how-Transfers und der Wissenskonservierung verloren. Zusätzlich wird die Wartbarkeit der Systeme durch projektproprietäre Änderungen an der Infrastruktur erschwert und mögliche Errungenschaften wie beispielsweise eine einheitliche Referenzarchitektur, erreicht durch konsequente Nutzung von Generation und Transformation, wieder zunichte gemacht.

Zuzuordnende Aufgaben

Zu den anfallenden Aufgaben, die entsprechenden Organisationseinheiten zugeordnet werden müssen, gehören:

- Die Erstellung und Pflege der Referenzarchitektur (Architekturmanagement),
- Management der Anforderungen an die MDA-Infrastruktur,
- Erstellung, Pflege und Weiterentwicklung (Evolution) der des Frameworks,
- die Durchführung von Kundenprojekten unter Benutzung der Infrastruktur,
- Prozessmanagement und -überwachung
- sowie der Erhalt der Zukunftsfähigkeit der Infrastruktur durch Evaluierung neuer Technologien und Trends und entsprechende Anpassung der Assets.

Ob diese Aufgaben bereits bestehenden Einheiten zugeordnet werden können, oder neue Strukturen geschaffen werden müssen, ist dabei im Einzelfall zu entscheiden. Als Entscheidungshilfe wollen uns nachfolgend einige Beispiele realer Organisationsstrukturen anschauen, die versuchen, den oben genannten Anforderungen gerecht zu werden.

6.5.1 Drei mögliche Organisationsmodelle

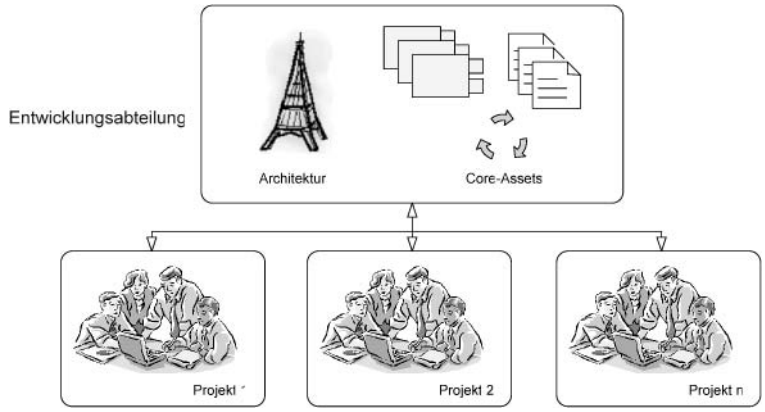
Die folgenden Betrachtungen basieren auf einer Studie von Jan Bosch, in der er eine Reihe von Unternehmen untersucht hat, die sich der Strategie der Software-Produktlinien verschrieben haben [Bos01] und sich auch im besonderen Maße für Strukturierung von Unternehmen mit geplanter Übernahme einer MDA-Strategie eignen:

Einzelne Entwicklungsabteilung

Abb. 6.5 zeigt eine Organisationsstruktur, in der die gesamte Softwareentwicklung in einer Einheit konzentriert ist. Die Pflege der MDA-Assets obliegt hier allen Entwicklern gemeinsam, was eine hohe Flexibilität und weitreichende Kenntnisse der Teammitglieder in allen MDA relevanten Bereichen voraussetzt.

Mitarbeiter (MA) sind dabei Projekten zugeordnet, die entweder die Entwicklung von Kundensystemen oder die Pflege und Wartung der Infrastruktur zur Aufgabe haben. Die Zuteilung der Mitarbeiter zu Projekten sowie die akut zu bewältigenden Aufgaben der Infrastrukturprojekte wechselt von Fall zu Fall.

Abb. 6.5 Einzelne Entwicklungsabteilung



Business Units

Mögliche Alternative beim Überschreiten der gerade erwähnten 30-MA-Grenze ist die Aufteilung der Infrastruktur-Verantwortung auf so genannte *Business Units (BU) (dt. Geschäftseinheiten).*

Während die initiale Erstellung der Infrastruktur oft Aufgabe aller Teams ist, sind die einzelnen Teammitglieder im Alltagsbetrieb

expliziten *Business Units* zugeordnet, denen wiederum einzelne Assets zur Wartung und Pflege fest zugeordnet sind. So kümmern sich Teams der einen Einheit beispielsweise um die Pflege des Prozessmodells, während Transformatoren und Generatoren einer anderen Einheit zugeteilt sind. Wenngleich also alle Gruppen benutzenden Zugriff haben, ist die Veränderung – insbesondere auch die Aufgabe der Evolution sowie Pflege einzelner Assets – explizit und ausschließlich bestimmten BUs zugeordnet. Die Vorteile dieses Ansatzes liegen in der besseren Skalierbarkeit (bis ungefähr 100 Personen) sowie in der besseren Kontrolle der Asset-Evolution.

Problem: Fehlende zentrale Kontrolle

Trotzdem fehlt immer noch eine zentrale Kontroll-/Eskalationsinstanz zur Steuerung der Evolution. Durch die fehlende Überwachung der Kohärenz zwischen den einzelnen Assets kommt es im Laufe der Zeit fast unweigerlich zu Erosionserscheinungen in der Gesamtarchitektur der Infrastruktur. Die Disziplin, die zur Verhinderung dieses Phänomens notwendig ist, kann ohne explizit vergebene Verantwortlichkeit ausschließlich aus der Unternehmenskultur selbst kommen, weshalb die Pflege derselben in einem solchen vordringliche Aufgabe des mittleren und höheren Managements ist.

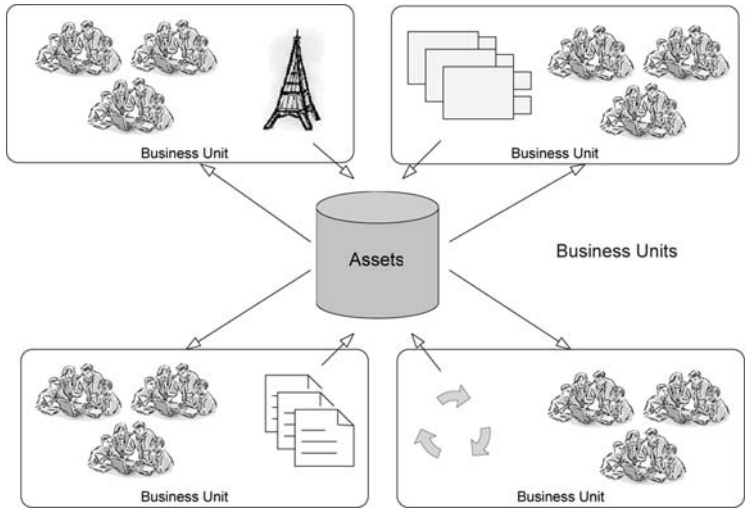


Abb. 6.6 Business Units

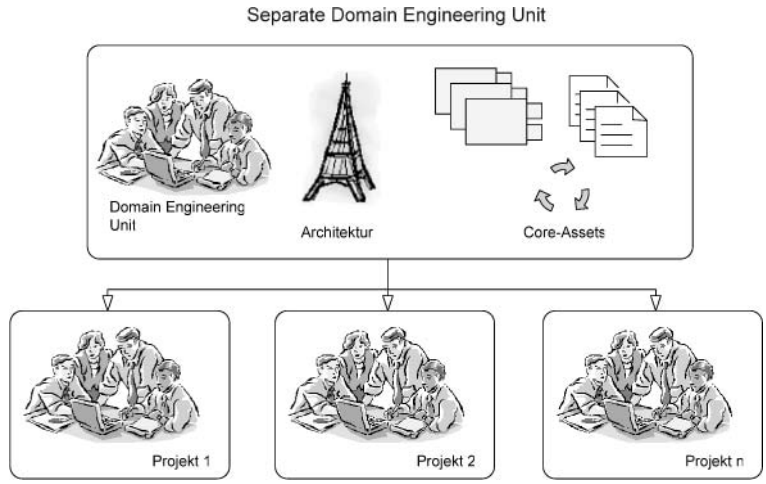
Domain Engineering Unit

Spätestens einer Größe von ca. 100 Mitarbeitern wird der Kommunikationsoverhead der eben beschriebenen Modelle so groß, dass eine zentrale koordinierende Instanz nicht länger vernachlässigt werden kann.

Abb. 6.7 zeigt ein Modell mit integrierter *Domain Engineering Unit (DEU)*, die genau diese Aufgaben wahrnimmt. Die Entwicklung von Kundenprojekten und die Pflege der Assets sind hier klar voneinander getrennt. Ein dezidiertes Team von Spezialisten kümmert sich hier ausschließlich um die alle Belange der Infrastruktur. Dargestellt ist hier die Existenz einer einzelnen DEU, denkbar ist jedoch auch die Aufspaltung in spezialisierte Untereinheiten (beispielsweise bei Anwachsen der DEU-Größe über 30 MA).

Größter Vorteil des Modells ist der Wegfall der notwendigen *n-zu-n* Kommunikation des Business Unit Ansatzes. Zusätzlich ist eine dezidierte DEU besser in der Lage den Gesamtüberblick über die Infrastruktur zu behalten und eine geeignete Evolutionsstrategie fest- und durchzusetzen. Eventuelle Interessenkonflikte, die zwischen Business Units auftreten können, sind besser durch eine solche unabhängige Instanz zu lösen. *Last-not-least* skaliert der Ansatz auch in größerem Maßstab.

Abb. 6.7 Domain Engineering Unit



Größte zu bewältigende Schwierigkeit ist die ständige und korrekte Erhebung der Anforderungen, die vonseiten der einzelnen Projektteams an die Infrastruktur bestehen.. Auch können durch das ausschließliche Recht der DEU, Assets zu modifizieren, Engpässe durch gleichzeitig auftretende Anforderungen, sowie dadurch bedingte Verlängerung der Time-to-Market einzelner Kundenprojekte entstehen. Ein Ausweg aus möglichen Engpass-Situationen ist das Zulassen temporärer, projektproprietärer Modifikationen von Assets durch Projektteams, die initial allerdings *nicht* der Allgemeinheit zur Verfügung gestellt werden. Hat die DEU letztlich Kapazitäten frei,

können diese Veränderungen überprüft, ggf. angepasst und nach erfolgreichen Tests und Freigabe durch die zuständige DEU-Instanz in den Bestand der allgemeinen Assets aufgenommen werden.

6.6 Best Practices und Gefährliches

An dieser Stelle sollen Gedanken, Methoden und Techniken, die in Realprojekten mit MDA-Ansatz und bei der Durchführung der bisher angeklungenen Aufgaben wertvolle Dienste leisten können angesprochen werden. Zudem soll auf Gefahren hingewiesen werden, die in MDA getriebenen Entwicklungsprojekten lauern. So liefert Abschnitt 6.6.1 nähere Einsichten in das Gebiet der iterativ-inkrementellen Softwareentwicklung, die dem gesamten vorgestellten Ansatz untrennbar unterliegt. Abschnitt 6.6.2 gibt Tipps, wie man es machen soll und Abschnitt 6.6.3 warnt davor, wie man es besser nicht macht.

6.6.1 Iterativ-Inkrementelle Softwareentwicklung

Von iterativer Softwareentwicklung spricht man dann, wenn der Entwicklungszyklus – die Zeit von Beginn der Entwicklung bis zum fertigen Endprodukt – aus einer Sequenz von *Iterationen* besteht. Aufeinanderfolgende Gruppen von Iterationen werden in *Phasen* aufgeteilt.

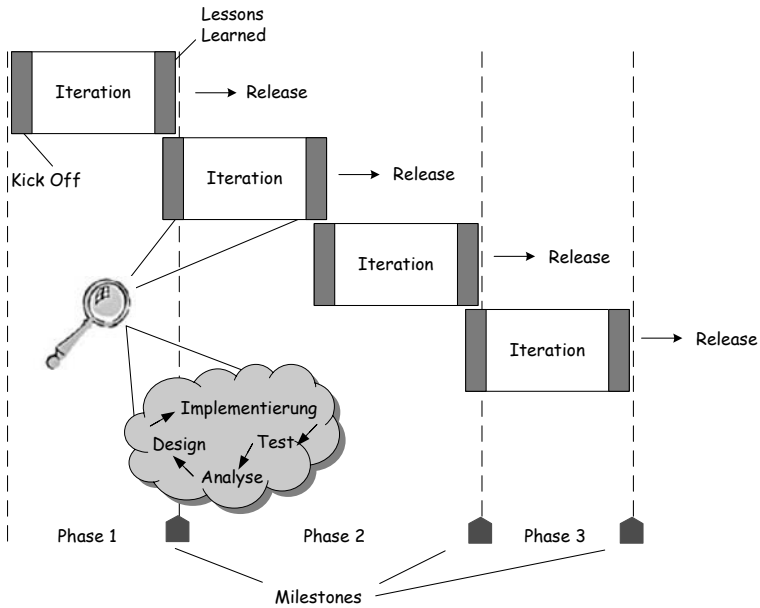


Abb. 6.8 Übersicht iterativ-inkrementelles Vorgehen

Eine einzelne Iteration kann als Miniprojekt aufgefasst werden, das alle Aktivitäten (z. B. Analyse, Design, Implementation, Testen, …) des Entwicklungsprozesses enthalten kann und an dessen Ende ein getestetes und integriertes (Software-) System steht, das Release genannt wird.

Mit jeder Iteration werden zusätzliche Anforderungen an das Endprodukt realisiert, der Umfang des Releases „wächst“. Aufgrund dieser schrittweisen Vorgehensweise bezeichnet man diese Art der Softwareentwicklung auch als *iterativ-inkrementelle Entwicklung (engl. iterativ incremental development, abgekürzt IID)*. In Abb. 6.8, die uns auch in den folgenden Abschnitten begleiten wird, sind einige dieser Konzepte graphisch dargestellt.

Aufbau von Iterationen

Eine Iteration beginnt mit einer *Kick-off-Phase*, in der dem Team der aktuelle Stand der Planung bekannt gemacht und die Ziele, zumindest der unmittelbar anstehenden Iteration, vorgestellt werden. Diese Planung wird direkt beeinflusst durch die Ergebnisse der Feedback-Aktivität (vgl. Abschnitt 6.2.6) am Ende der vorangegangenen Iteration (*Lessons- learned*).

Zusammenspiel von IID und MDA

Bevor wir uns weitere Details der IID anschauen, wollen wir kurz die gerade beschriebenen Konzepte auf ihre Eignung hinsichtlich der Anwendung im MDA-Kontext untersuchen.

Kernkonzept von MDA ist die Annahme, dass komplexe Systeme nur dann beherrscht werden und die in sie investierte Arbeit geeignet wiederverwendet werden können, wenn sie in ihre Teilaspekte zerlegt werden. Konkret bedeutet dies die Erstellung und schrittweise mit dem Verständnis des Systems einhergehende Verfeinerung der Modelle, die diese Aspekte beschreiben und dann zur automatischen Erzeugung der Applikation genutzt werden. Betrachtet man die Schritte, die zur Ausführung einer solchen Vorgehensweise notwendig sind, wird die natürliche Affinität von MDA und IID deutlich: Modelle spiegeln das Verständnis des Systems wider, das das Entwicklungsteam zu einem bestimmten Zeitpunkt erworben hat. Gerade in der Anfangsphase sind diese Modelle daher keinesfalls perfekt. Sowohl fachliche Fehler als auch Fehler im Design der Programmarchitektur machen eine erneute Analyse sowie kontinuierliches Refactoring bzw. Redesign der bestehenden Sichten (der Modelle) des Systems notwendig.

Im Wesentlichen ist die Kernaussage: Alle erstellten Artefakte (Anforderungsdokumente, Modelle, …) werden nicht „so schnell wie möglich fertiggestellt und reviewed“, sondern auf einen Stand

gebracht, der dem aktuellen Wissen des Entwicklungsteams entspricht. Wächst dieses Wissen im Verlauf des Projektes oder werden Änderungen nötig, da sich beispielsweise Kundenwünsche ändern, werden diese Artefakte inkrementell im Laufe des Gesamtprojektes vervollständigt bzw. angepasst.

Evolutionary/ Adaptive Development

Manche Autoren benutzen für die Beschreibung einer solchen Vorgehensweise auch den Begriff der *Evolutionären Entwicklung*, der den Aspekt der schrittweisen Anpassung des Systems an das gewünschte Endergebnis hervorhebt. Synonym wird diese Art der Entwicklung auch *Adaptive Entwicklung* genannt. Unterstützt wird das Vorgehen durch das iterative Konzept. Nach jeder Iteration wird die Applikation aus dem Modellbestand generiert und, komplettiert mit manuell erzeugten Artefakten, die Anwendung integriert. Dieses Release wird dann intern oder unter Einbeziehung des Kunden auf fachliche wie auch auf technische Korrektheit geprüft.

Auswahl der Iterationsziele

Die Auswahl der Anforderungen, die im nächsten Release zur Implementierung anstehen, sollte im Wesentlichen durch zwei Faktoren beeinflusst werden:

- Features, deren Realisierung mit einem gewissen Risiko behaftet ist, sind möglichst früh im Projektwerdegang anzugehen, um gegebenenfalls alternative Lösungswege einschlagen zu können, bzw. bei Erkennen der Unmöglichkeit des Vorhabens das Projekt abzubrechen und damit den finanziellen Schaden zu begrenzen. Typisches Beispiel für ein solches Szenario sind Probleme bei der Anwendung neuer Technologien bzw. fehlende Erfahrung des Entwicklerteams im Einsatz derselben. Softwareentwicklung, die diesen Faktor beachtet, wird *risiko-getriebene Entwicklung (engl. risk-driven development)* genannt.

  Risk-driven Development

- Die Software soll möglichst früh an den Kunden ausgeliefert werden. Hier sind vor allem die Teilaspekte des Endproduktes interessant, die bereits sinnvoll eingesetzt werden können. Geht man auf diese Weise an die Planung heran, werden die Ziele des nächsten Releases also vom Kunden gesetzt. Man spricht daher von *kunden-getriebener Entwicklung (engl. customer-driven development).*

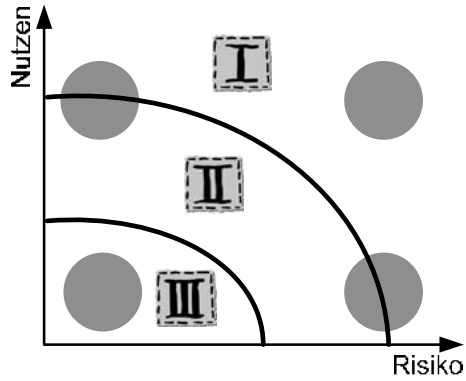
  Customer-driven Development

Die Beachtung beider Prinzipien ist erstrebenswert; die daraus resultierenden, unterschiedlichen Ziele lassen sich jedoch in den seltensten Fällen 100%ig miteinander in Deckung bringen. Die meisten Vorgehensweisen versuchen daher einen Kompromiss zu

finden, der beide Aspekte in Betracht zieht. Der unterschiedlichen Auffassung von Kunden und Entwicklern über das, was möglichst schnell in Angriff zu nehmen ist, muss dabei unbedingt Rechnung getragen werden. Projektmanager mit technischem Background (z. B. frühere Entwickler) tendieren dazu, möglichst schnell alle Features zu realisieren, die ihnen technologisch am anspruchsvollsten erscheinen, um Risiken und Verluste zu minimieren. Der Kunde als Geldgeber wiederum hätte gerne möglichst schnell die Features mit dem für ihn größten Nutzen.

Abb. 6.9 zeigt auch grafisch, dass diese Ansprüche nicht immer zu gleichen Teilen befriedigt werden können. Anwendungsfälle mit hohem Nutzen und geringem Risiko (Bereich oben links) stehen Fällen mit hohem Risiko aber relativ geringem (funktionalen) Nutzen gegenüber (Bereich unten rechts). Die Aufgabe ist nun, einen vernünftigen Kompromiss zwischen beiden Interessen zu finden, wenn man die zuerst zu realisierenden Anwendungsfälle auswählt. Teilt man das Diagramm in drei Cluster (`I-III`) auf, so bestünde eine Daumenregel zur Erstellung einer guten „Mischung"' der beiden Interessen in dem Ratschlag, sich bei der Auswahl der zu erreichenden Ziele am Anfang des Projektes auf Anwendungsfälle der Cluster `I` und `II` zu beschränken.

Abb. 6.9
Risiko-Nutzen Cluster



Planung der Iterationsdauer durch Timeboxing

Keep delivering

Die Länge der Iterationen innerhalb eines Projektes ist in vielen Prozessmodellen variabel gehalten und sollte sich in der Praxis zwischen 2 - 6 Wochen bewegen. Eine Technik, die die Planung der Dauer individueller Iterationen unterstützt, ist das *Timeboxing*. Einmal festgelegt, ist bei angewandtem Timeboxing die Dauer einer Iteration als fest und unumstößlich zu betrachten. Am Fälligkeitsdatum ist auf jeden Fall ein Release fertig zu stellen. Dies können auch interne Releases sein, die zwar nicht an den Kunden weitergereicht,

dafür jedoch zwingend intern getestet und qualitativ bewertet werden, um das weitere Vorgehen festzulegen.

Eine Sicht, die nochmals den gesamten Entwicklungszyklus eines IID-Projektes verdeutlicht, bietet Abb. 6.10. Die Darstellung betrachtet ein Projekt als eine Verschachtelung zyklischer Prozesse (engl. nested cyclic processes) und stammt aus der Beschreibung von Alistair Cockburns Crystal Method [Coc04], in der diese Sichtweise näher erläutert und motiviert wird. Das Timeboxing findet in dieser Darstellung in der Begrenzung der Zyklenlänge einer einzelnen Iteration statt.

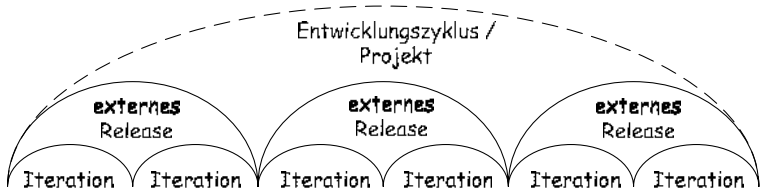


Abb. 6.10 Entwicklungszyklus (nach [Coc04])

Eskalation bei Zeitverzug

Downscoping

Wird im Verlaufe der Iteration deutlich, dass nicht alle ursprünglichen Anforderungen rechtzeitig erfüllt werden können, wird die Zahl der Ziele soweit verringert (engl. *Downscoping*), dass am Ende der Iteration trotzdem ein stabiles, integriertes, getestetes System vorliegt. Timeboxing darf jedoch nicht mit der Erhöhung des Drucks auf die Entwickler verwechselt werden. Wie spätestens seit *Peopleware* [DL99] in das Bewusstsein aller Projektleiter eingedrungen sein sollte, darf die Antwort auf drohenden Zeitverzug nicht in der dauerhaften Anwendung der Überstunden- und Mehrarbeitskeule (z. B. an Wochenenden) liegen, sondern ausschließlich in der Anwendung konsequenten Downscopings bzw. der neuerlichen Betrachtung des magischen Zieldreiecks (also der Anpassung einer der Größen: Funktionsumfang, Dauer oder Aufwand/Kosten).

Zur Umsetzung dieser Vorgehensweise ist es sinnvoll, Anforderungen in Primär- und Sekundärziele aufzuteilen. Primärziele sollten realistisch eingeschätzt werden und auf jeden Fall erreichbar sein. Sekundärziele stellen den Pool von Zielen dar, aus dem im Falle eines Zeitverzugs Features ausgewählt werden, die für spätere Iterationen zurückgestellt bzw. bei Zeitgewinn Ziele vorgezogen werden können. Wichtig ist hervorzustreichen, dass die Änderung von Iterationszielen nur vom Entwicklerteam durchgeführt werden darf, das direkt an der Realisierung beteiligt ist. Verhindert werden soll damit eine Zersetzung der Zeitplanung durch den sogenannten

creeping featurism (engl.) [CF], der durch den Satz *„Schön wäre auch, wenn wir Feature XY noch kurz mitrealisieren könnten.“* gut beschrieben wird.

Timeboxing in MDA-Projekten

Bei Verwendung des Timeboxing-Konzepts ist am Ende einer Iteration immer eine Integration des Gesamtsystems vorzunehmen. Dies bedeutet im MDA-Kontext eine hohe Voraussetzung von Disziplin an die Projektbeteiligten, die noch höher sein kann, als beim herkömmlichen Vorgehen. Diese Konsequenz ergibt sich aus der Tatsache, dass große Teile des Systems aus Modellen generiert werden. Der automatisierte Durchlauf durch die Modellkette funktioniert notwendigerweise nur dann, wenn alle Modelle am Ende der Iteration in konsistentem, formal korrektem Zustand vorliegen. Ist nur ein Glied in der Generierungs-/Integrationskette fehlerhaft, findet die Arbeit der am unteren Strom Arbeitenden unweigerlich in einer oder mehreren Fehlermeldungen ihr jähes Ende.

Continuous Integration

Obwohl Schutzmaßnahmen wie die Benutzung von Versionskontrollmechanismen hier eine gewisse Sicherheit bieten können, sollten alle Projektteilnehmer konkret darauf sensibilisiert werden, dass in generativen Ansätzen im Extremfall schon ein einzelner Fehler in der Kette die Gesamtintegration verhindern kann. Best Practice, die das zu späte Integrieren und die damit verbundenen Probleme direkt angeht, ist die *Continuous Integration*. Hier wird automatisiert in kurzen Zeitintervallen die Integration des Gesamtsystems vorgenommen (Build-Prozess), um Inkonsistenzen frühzeitig zu erkennen. Ein bekanntes Open Source Tool hierfür, das unter anderem die Benachrichtigung der betroffenen Entwickler sowie eine webbasierte Übersicht bietet, ist CruiseControl, das unter [CC] auf Sourceforge gehostet wird. Allgemeine Gedanken zu Continuous Integration von Martin Fowler und Matthew Foemmel finden sich unter [CI]. Generell ist die Bedeutung eines funktionierenden und automatisierten Build-Prozesses mit integriertem Systemtest in größeren (MDA-) Projekten nicht zu unterschätzen. Hier sollte einiges an Ressourcen und Gehirnschmalz investiert werden, um bösen Überraschungen im Verlauf des Projektes vorzubeugen (siehe dazu auch Fallstrick *Phlegmatischer Build-Prozess* in Abschnitt 6.6.3).

Grobplanung des Gesamtprojekts durch Milestones

Kommen wir noch einmal zur Planbarkeit von IID-Projekten zurück: Gewöhnlich werden Iterationsziele erst kurz vor Beginn der Iteration festgelegt (mit einer Vorlaufzeit von 1-2 Iterationen) oder können aufgrund von Ergebnissen der Lessons-learned Aktivität kurzfristig noch umgeworfen werden. Bei Anwendung des Downscoping-Konzeptes kann dies sogar während der laufenden Iteration durch das entwickelnde Team selbst geschehen.

Die Frage, wie ein Projekt überhaupt planbar gemacht werden kann, das aus Iterationen besteht, deren konkrete Dauer bei Projektbeginn nicht feststeht, fordert aller Flexibilität zum Trotz eine klare Antwort. Das Prinzip, das an dieser Stelle aushilft, besteht in der Schaffung eines Projektrahmens, der das Projekt in Phasen einteilt und den Zeitpunkt enthält, wann zur nächsten Phase übergegangen wird sowie die Anforderungen stellt, die vor Übergang zu dieser Phase erfüllt sein müssen, enthält. (Abb. 6.11).

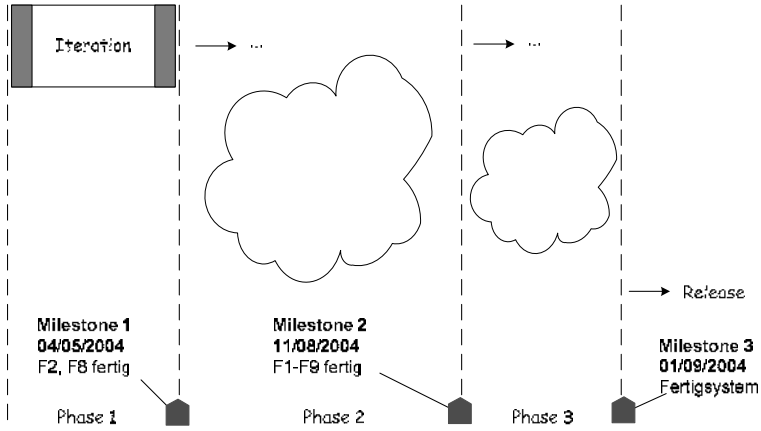


Abb. 6.11 Adaptive Planung und Milestones

Der Rahmen wird bereits in der Planungsphase des Projektes gespannt, indem *Milestones (dt. Meilensteine)* definiert und deren Fälligkeitsdaten festgelegt werden. An jeden Milestone sind Bedingungen geknüpft, die erreicht sein müssen, damit das Projekt in die nächste Phase gelangen kann. Diese Bedingungen können beispielsweise Listen von Anforderungen sein, die zum Stichtag in der Anwendung realisiert sein müssen. Werden eines oder mehrere Kriterien nicht erfüllt, tritt ein Risikoplan in Kraft, oder das Projekt wird zur Abwendung weiteren (finanziellen) Schadens abgebrochen. Auch die Beschreibung von Mechanismen zur Behandlung dieser Ausnahmefällen, in Form eines Risikoplans als Teil eines dezidier-

ten Risiko-Managements, muss Teil der Anfangsplanung eines nach State-of-the-Art durchgeführten Entwicklungsprojektes sein.

Adaptive Planung

Die Abbildung (Abb. 6.11) zeigt einen Schnappschuss der Projektplanung zu einem Zeitpunkt innerhalb der ersten Iteration. Während die Daten der Milestones sowie die Kriterien, die zum erfolgreichen Erreichen derselben notwendig sind – in Form von Feature Nummern *Fx* – schon feststehen, ist die Anzahl der Iterationen nur für die erste Phase vorgeplant (1 Iteration). Dieses adaptive Planung oder *rolling wave planning (engl.)* genannte Vorgehen bietet Planungssicherheit und Projektkontrolle in großen Zeiträumen sowie gleichzeitige Gewährleistung eines erheblichen Maßes an Flexibilität auf kurze Sicht und ermöglicht so die schnelle Reaktion auf kritische Situationen im Projektalltag.

Standard Milestones nach Barry Boehm

Wer sich näher mit dem Konzept der Milestones auseinander setzen möchte, findet im Artikel *Anchoring the Software Process* [Boe95] von Barry Boehm eine erste Anlaufstelle. Dort werden drei „Standard Milestones“ vorgeschlagen:

- *Life Cycle Objectives (LCO)*: Top-Level Anforderungen, erstes Realisierungs- und Architekturkonzept, Machbarkeitsanalyse, Stakeholder, erste Projektplanung
- *Life Cycle Architecture (LCA)*: Ausarbeitung der Artefakte des ersten Milestones
- *Initial Operational Capability (IOC):* Lauffähiges System, Schulung der Kunden und Maintainer, Auslieferung

Dies bedeutet nicht, dass Boehm nur insgesamt drei Milestones pro Projekt vorschlägt. Zwischen bzw. nach den drei vorgeschlagenen können beliebig viele selbstdefinierte Milestones existieren bzw. eingefügt werden. Auch erwähnenswert ist, dass die Verwendung der drei „Standard Milestones“ keine Notwendigkeit zur Adaption eines bestimmten Prozessmodelltyps impliziert. Sie lassen sich ebenso in ein Wasserfallmodell wie in einen iterativen Kontext einbinden, sind also in diesem Sinne generisch. Welches der beiden Vorgehensmodelle wir empfehlen würden sollte jedoch im Verlaufe des Kapitels deutlich geworden sein …

6.6.2 Best Practices …

An dieser Stelle folgt die Erörterung einer mehr oder minder losen Sammlung von uns empfohlener Vorgehensweisen und Techniken, die sich im Einsatz vieler Projekte als besonders geeignet für die

Durchführung modellgetriebener Softwareentwicklung hervorgetan haben. Hierbei handelt es sich um Techniken, die orthogonal zum verwendeten Prozessmodell stehen und damit so etwas wie „Allzweckwaffen“ darstellen.

Technologien realistisch einschätzen

Schätzen gehört zum Projektalltag. Fehlende Erfahrungen mit den einzusetzenden Technologien oder fehlende Referenzprojekte, die den Verantwortlichen die realistische Einschätzung der Komplexität und damit von der realistischen Dauer eines Projektes erleichtern könnten, machen viele Projekte jedoch schon vor deren Beginn zu gescheiterten Unternehmungen, da die Vorgaben schlicht nicht einzuhalten sind. Wie bei jeder „gehypten“ Technologie werden auch im MDA-Kontext zu oft in großem Maße übertriebene Werbebotschaften von möglichen Zeitersparnissen und Effizienzsteigerungen verbreitet, die, erst einmal ins Unterbewusstsein eingedrungen, in unrealistischen Planungen und damit im Endeffekt in einem gescheiterten Projektzeitrahmen enden.

Hype und Realität

Der Artikel *The Realities of Software Technology Payoffs* von Robert L. Glass [Gla99] bietet hier Ernüchterung sowie einen nicht minder wichtigen harten Wasserstand, was die tatsächliche Effizienzsteigerung von so genannten *Breakthrough Technologien* angeht. Die Angabe von Steigerungen (in Form von Zeitersparnis) mit Durchschnittswerten im Bereich von lediglich 5, bis höchstens 35% sollten stutzig und vorsichtig machen bevor allzu übersteigerte Hoffnungen in diesem Bereich in den MDA-Ansatz gelegt werden. Übersteigerte Erwartungen produzieren vor allen Dingen eins: Enttäuschung! Was zur Folge hat, dass oftmals der gesamte Ansatz verteufelt wird und – oft zu Unrecht – in der Ecke der gescheiterten und damit für alle Zeiten verbrannten Ansätze landet.

Runaway Projects

Dass die falsche Einschätzung der Komplexität neuer Technologien keine rein theoretische Gefahr darstellt, belegt eine Studie von Andy Cole aus dem Jahr 1995 [Col95]. Unter dem Titel *Runaway Projects* wurden dort Projekte untersucht, die:

- ihre Vorgaben/Ziele klar verfehlt haben oder
- eine Budgetüberschreitung von Zeit oder finanziellen Mitteln von über 30% aufwiesen.

Über die Hälfte der befragten Unternehmen hatten Erfahrungen mit mehr als vier Projekten in der Größenordnung über 500.000 Euro

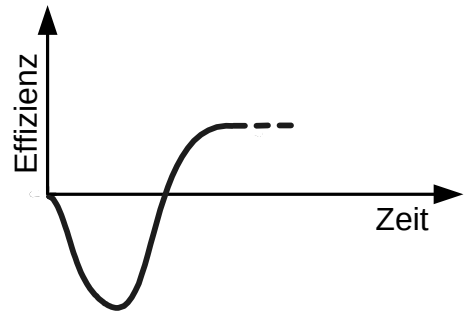
gesammelt. Über 60% der Teilnehmenden hatten dabei schon ein Runaway Project Erlebnis hinter sich.

Als Hauptursachen (angegebene Gründe in mindestens 50% der Projekte) kristallisierten sich Probleme im Management der Anforderungen, sowie Fehleinschätzungen von Technologien heraus. Interessant ist die Ansicht eines großen Prozentsatzes der Befragten, dass sie die verwendete Technologie in weiteren Projekten wieder einsetzen würden. Die Schuld wird also klar in Fehlern innerhalb des Managements im Vorfeld identifiziert und nicht etwa auf die benutzen Werkzeuge o. Ä. geschoben.

Prämisse im Vorfeld einer geplanten MDA-Einführung muss also sein, dass Effizienzsteigerungen durch MDA-basierte Ansätze realistisch eingeschätzt werden müssen. Lernkurven bei den ersten Gehversuchen müssen mit einkalkuliert werden, um so Enttäuschungen zu vermeiden und damit die Motivation der Projektbeteiligten auch in Folgeeinsätzen zu erhalten.

Abb. 6.12 zeigt den Verlauf dieser für neue Technologien charakteristischen Lernkurve, die es zu überwinden gilt, um sie Gewinn bringend einzusetzen.

Abb. 6.12 Lernkurve beim Einsatz neuer Technologien



Kennzeichnend ist, dass die Kurve zwar konzeptionell gezeichnet, die Skalen jedoch nicht sinnvoll angegeben werden können. Sowohl die nötige Zeit, als auch der zu erwartende Benefit hängen stark von den individuellen Fähigkeiten des Teams, das mit dem Ansatz arbeiten soll, sowie im selben Maße von der aktuellen Projektsituation ab. Allgemeinplätze wie „Unser Produkt erhöht die Leistungsfähigkeit ihrer Entwickler um bis zu 800%!“ sind also nichts als leere Versprechungen.

Anwendung offener Technologien

Wie die meisten bis zu diesem Zeitpunkt erwähnten Prinzipien, ist die Maxime sich so weit wie möglich auf offene Technologien zu stützen und proprietäre (d. h. herstellergebundene) Komponenten nur dann zu verwenden, wenn sich keine vernünftige Alternative bietet, kein spezielles Anliegen der MDA.

Aufgrund des geringen Alters des MDA-Ansatzes und des ihn umgebenden „Hypes“ ist jedoch besondere Vorsicht geboten (vgl. dazu auch die die Ausführungen zu *Shelfware* in Abschnitt 6.6.3). Die Standards der OMG sind zurzeit noch einer hohen Instabilität unterworfen und haben zum Teil zur Drucklegung dieses Buches noch nicht den finalen Status erreicht. Und auch nach Fertigstellung der Standards ist immer noch nicht garantiert, dass sie von der Industrie auch konsequent umgesetzt werden. Beispiel: Die Austauschbarkeit von UML-Modellen der Versionsreihe 1.x lässt zwischen Tools vieler Hersteller noch viele Wünsche offen, obwohl die Möglichkeiten längst gegeben sind. Ein Schelm ist derjenige, der dabei mehr Methode als technische Probleme vermutet.

Zum Glück existieren jedoch vielfältige Alternativen zu teurer Kaufsoftware. Da gerade im Bereich der Open-Source Software (OSS) in nahezu allen Bereichen kostenlose, offene Lösungen angeboten werden, ist die Gleichung:

```
MDA = Kaufen einer teuren Komplettlösung
```

glücklicherweise also trotzdem eine Falsche.

Das Warten auf den Patch

Neben den monetären Gründen ist die Anwendung von OSS auch aufgrund der besseren Integration in die individuelle Entwicklungslandschaft der eigenen Firma vorteilhaft. Während kleine Änderungen in offenen Produkten oft selbst gemacht werden können bzw. die Änderungszyklen und Releases im OSS-Bereich oft viel kürzer sind als im kommerziellen Sektor, ist man beim Kauf eines großen Software-Paketes auf den (oft nicht minder teuren) Support bzw. auf das lange Warten auf einen Patch angewiesen, der vielleicht nie kommt. Auch überschreitet die Komplexität vieler Komplettlösungen, die notwendig werdenden Anpassungen und die Schulung von Mitarbeitern auf genau dieses Produkt, leicht den zu erwartenden Gewinn an Effizienz.

Auch bei OSS ist Vorsicht geboten

Zu beachten ist jedoch, dass im OSS-Bereich zwar viele hoffnungstragende Projekte begonnen, aber mindestens ebenso viele abrupt und endgültig wieder beendet werden. Bevor man sich also einer

OSS-Technologie voll und ganz verschreibt, sollte man daher darauf achten, dass:

- noch Leben in den Aktivitäten des Projektes steckt (z. B. messbar in der Häufigkeit neuer Releases, Beiträgen in zugehörigen Foren usw.),
- ein ausreichender Support über Mailinglisten und Newsgroups gewährleistet wird und
- das Projekt bereits einen bestimmten Bekanntheitsgrad in der Software-Gemeinde erreicht hat. Zu erkennen ist dies z. B. an öffentlich zugänglicher Dokumentation von Nicht-Projektangehörigen sowie verwandter Buchpublikationen, Schulungen, Online-Diskussionen o. Ä.

Paradebeispiele für solche Projekte sind das *Jakarta Project* [Jakarta], die Entwicklungsumgebung *Eclipse* [Eclipse] von IBM, als auch *JBoss* [JBoss] als OSS Implementierung des JavaEE Standards.

Der notwendige Pragmatismus oder „Agile MDA"

Abschließend soll in diesem Abschnitt noch auf einen Punkt eingegangen werden, der sowohl bei der ersten Idee MDA auszuprobieren bis hin zur Anwendung im konkreten Projekt eine Hauptrolle spielen sollte: Bei aller MDA darf der notwendige Pragmatismus, bzw. die Erhaltung der Projekt-Agilität nicht vergessen werden.

Für einige Kritiker ist dies bereits ein Widerspruch in sich. Agilität und MDA? Das kann schon deshalb nicht zusammenpassen, weil bereits das „M" in MDA etwas verlangt, das mit Agilität nicht vereinbar scheint: das Erstellen und die Pflege formaler Modelle während des gesamten Entwicklungszyklus. In seinem Buch *Agile Software-Development* [Coc01] gibt Alistair Cockburn auf solche Vorbehalte eine klare Antwort:

> *Die Frage […] lautet nicht „Kann in dieser Situation eine agile Methodik verwendet werden?", sondern „Wie schaffen wir es, in dieser Situation agil zu bleiben?".*

In unserem Falle bedeutet das: „Wie können Modelle so in den Entwicklungszyklus eingebunden werden, dass ihre Erstellung nicht zum Selbstzweck degeneriert?" Die Antwort darauf ist schon in den vorangegangenen Abschnitten angeklungen. Fernziel ist es, die Modelle ausführbar zu machen. Dadurch werden sie selbst zum Endprodukt und die Diskussion über ihre Notwendigkeit erledigt sich von selbst. Ansätze zum Erreichen dieses Ziels sind bereits auf

den Weg gebracht. Bevor es jedoch soweit ist – wenn es überhaupt erstrebenswert sein sollte, komplett auf Quellcode zu verzichten – muss darauf geachtet werden, dass alle erstellten Modelle entweder als Dokumentation oder als Ausgangspunkt zur Transformation bzw. Codegenerierung wirklich benötigt werden.

Keine Weltmodelle!

Dadurch, dass Modelle im MDA-Ansatz einen hohen Stellenwert einnehmen, sind die Beteiligten nur zu oft versucht zuviel zu modellieren. Auch hier gilt: Es sollte immer nur das modelliert werden, was auch gefordert wurde. Bauen Sie keine Weltmodelle!

Bei einer Generierung von Quellcode ist die Einsparung manueller Programmiertätigkeit, bzw. die Wiederverwendbarkeit der Modelle ein Anhaltspunkt, der zur Entscheidung *Erstellung ja oder nein* herangezogen werden kann. Die Wichtigkeit von Modellen zu Dokumentationszwecken kann unter zwei Gesichtspunkten bewertet werden:

Dokumentation ist wichtig

Naheliegend ist die Anforderung des Kundens nach dokumentierenden Modellen als Endprodukt. Als Beispiel können hier Geschäftsprozessmodelle dienen. Unter denselben Aspekt fallen auch Modelle, die die Kommunikation zwischen Entwicklern und Kunde verbessern. Korrekt zu diesem Zwecke erstellte Modelle befinden sich auf einer höheren Abstraktionsebene als beispielsweise Quellcode und bedienen sich der oft nicht technischen Sprache des Kunden. In Situationen in denen komplexe Geschäftsabläufe innerhalb der Software berücksichtigt werden müssen kann die Modellierung dieser Abläufe ein eleganter Weg zur Ausräumung von Missverständnissen sein. Durch die relativ hohe Stabilität der modellierten Sachverhalte in diesem Bereich sind auch die den Sachverhalt darstellenden Modelle selbst oftmals über den gesamten Entwicklungszyklus stabil und behalten dadurch auch nach Abschluss des Projektes ihren Wert.

Das Software-Spiel

Den Denkanstoß zum zweiten Gesichtspunkt, der die Erstellung von Modellen bzw. Dokumentation auch unter agil-pragmatischer Sichtweise als erstrebenswert erscheinen lässt, liefert wiederum Cockburn in [Coc01], indem er Softwareentwicklung mit einem Spiel vergleicht, das die folgenden Ziele verfolgt:

> *Das primäre Ziel ist es, Software zu liefern; das Zweite ist es, alles für das folgende Spiel vorzubereiten.*

Wie das erste Ziel erreicht werden kann, liegt auf der Hand: Falls Sie keine Software liefern, ist es völlig unerheblich, wie nett Sie

alles für das nächste Spiel vorbereitet haben. Falls Sie andererseits eine Software geliefert haben und lediglich eine dürftige Vorbereitung für das nächste Spiel geliefert haben, gefährden Sie das Gesamtziel.

Der komplette Verzicht auf Dokumentation belässt jegliches Wissen über das erstellte System sowie über den beschrittenen Lösungsweg bei den am Projekt beteiligten Entwicklern. Stehen diese für Folgeprojekte nicht mehr zur Verfügung, ist ihre erfolgreiche Durchführung und die Gewährleistung der Pflege und Wartung des Systems an sich gefährdet.

Wieviel MDA ist noch pragmatisch?

Die Entscheidung darüber, wie viele Modelle innerhalb einer pragmatischen Herangehensweise an die MDA notwendig sind oder nicht, ist also von der jeweiligen Projektsituation sowie des geplanten Umgangs mit dem System nach Abschluss des Projektes abhängig:

- Sind Folgeprojekte geplant?
- Ist in der Durchführung des Projektes Wissen enthalten, das projektübergreifend einsetzbar ist und daher in geeigneter Form, d. h. im günstigsten Fall automatisiert und multiplikativ wiederverwendbar, konserviert werden soll?
- Ist das Projekt so groß, dass allein zur Kommunikation zwischen den Beteiligten die Erstellung von Dokumentation unverzichtbar ist?
- Stellt die leichte Wartbarkeit des Systems eine Anforderung an die zu erstellende Anwendung dar?
- Ist das System in einen komplexen Kontext von Legacy Anwendungen und Geschäftsprozessen eingebunden, die alle berücksichtigt werden müssen?
- …

Sind diese Fragen klar mit *„Nein.“* zu beantworten, kann die Anwendung eines MDA-Ansatzes innerhalb des Projektes sicherlich auf ein Minimum reduziert bzw. die Anwendung an sich überhaupt infrage gestellt werden. Bei Beantwortung einer dieser Fragen mit *„Ja.“* ist die Abwägung der erwähnten Interessen notwendig.

Dieser Abschnitt soll Sie vor allem für eines sensibilisieren: Vor die Entscheidung gestellt einen MDA-basierten Ansatz innerhalb eines Projektes zu verwenden ist auf jeden Fall zu überprüfen, ob MDA überhaupt zusätzlichen Nutzen bringen kann und wenn, in welchem

Maße die Anwendung der Prinzipien sinnvoll erscheint. Pragmatisch heißt nutzbringend. Dies gilt natürlich auch für die Frage ob MDA überhaupt für die aktuellen Anforderungen geeignet ist.

Es muss nicht immer UML sein

Obwohl UML die Sprache der OMG ist, sollte ihre Verwendung nicht zum Selbstzweck degenerieren und bei der Auswahl der verwendeten Modellierungssprachen immer nur der tatsächliche Nutzen und die Eignung der Sprache für die geplante Verwendung ausschlaggebend sein. So ist beispielsweise trefflich darüber zu streiten, ob graphische Modelle dafür geeignet sind Quellcode, zur Gänze zu ersetzen. Auch für die Darstellung anderer Aspekte, wie z. B. Abfragen, Geschäftsregeln usw., ist oftmals die Verwendung textueller DSLs den graphischen Pendants vorzuziehen.

Eventuell existiert momentan ja auch weder eine graphische noch eine textuelle Sprache, die ihre Ansprüche erfüllt: Bauen Sie Ihre eigene! Wie wir noch im praktischen Teil sehen werden, ist es nicht aufwendig,einen eigenen (*einfachen*) Editor für solche Modelle/Diagramme zu bauen bzw. Parser für textuelle Sprachen zu realisieren. Fazit: Benutzen sie die UML nur dort, wo es Sinn macht. Weichen sie ggf. auf andere (Modellierungs-)Sprachen aus. Modellierung macht nur dann Sinn, wenn sie als Werkzeug zur Vereinfachung des Entwicklungsprozesses dient. Die Wahl der optimalen Sprachen zur Darstellung der betrachteten Aspekte ist wesentliche Grundvoraussetzung zur Erfüllung der gesteckten Ziele.

6.6.3 … und Gefährliches

Nachdem wir im vorigen Abschnitt gesehen haben, wie man das Risko des Auftretens von Problemen in MDA-Projekten verkleinern kann, wollen wir nun auch die „dunkle Seite“ näher anschauen und eine Auswahl „Bad Practices“ vorstellen, die nahezu immer zu Problemen führen und im eigenen Projekt vermieden werden sollten. Wie in diesen dunklen Gefilden üblich, wird mit überspitzten Formulierungen nicht gegeizt, um die Kerngedanken hinter der Warnung deutlich werden zu lassen.

Entscheidungen ohne Konsequenz

Bisher haben wir Sie davor gewarnt, sich überstürzt in das „Abenteuer MDA“ zu werfen, ohne vorher das Für und Wider einer solchen Entscheidung abzuwägen. An dieser Stelle steht die Warnung vor dem Umkehrschluss: Haben Sie sich einmal für die Einführung der MDA entschieden, seien Sie unbedingt konsequent und vermeiden Sie zu zögerliches Vorgehen! Kommunizieren Sie ihren Ent-

schluss den betroffenen Mitarbeitern und zeigen Sie bzw. das Management, das die Entscheidung zu verantworten hat, unbedingten Rückhalt gegenüber den geplanten Maßnahmen. Nur so erreichen Sie, dass das Vorhaben auch wirklich ernst genommen wird und damit die notwendige Unterstützung im Unternehmen erhält.

Rückhalt bedeutet aber auch finanzielle und personelle Unterstützung. Stellen Sie den Durchführenden sowohl die geld- und zeitlichen Mittel, die benötigt werden zur Verfügung, stellen Sie aber auch sicher – und diese Komponente ist ebenso wichtig – dass die benötigten personellen Ressourcen vorgehalten werden. Dies bedeutet, dass gerade Projekten mit pilotierendem Charakter Ihre wirklich guten Leute dauerhaft zugeordnet und nicht beim nächsten gewonnenen Neuprojekt wieder herausgezogen werden (vgl. hierzu auch die Ausführungen zu *Pilotprojekten* in Abschnitt 6.4.2).

Kleiner Aufwand, k(l)eine Resultate. Diese Warnung klang schon einmal in diesem Kapitel durch und stellt eine nicht zu unterschätzende Gefahr dar. Bevor sie halbherzig an die Sache herangehen, lassen Sie es lieber sein. Oft hat man vor den Augen des Kunden, des Managements und der eigenen Mannschaft nur den einen Versuch eine neue Vorgehensweise zu etablieren. Scheitert dieser an mangelnder Konsequenz gehört sie fortan zur „verbrannten Erde" der gescheiterten Vorhaben und verschwindet damit erst einmal für unbestimmte Zeit in der Versenkung. Tritt dieser Worst-Case ein, leiden alle Beteiligten: Der Kunde verliert das Vertrauen in ihre Entscheidungs- und Realisierungskompetenz, das Management wird beim nächsten Versuch noch vorsichtiger sein die benötigten Mittel zur Verfügung zu stellen und die Motivation der eigenen Mannschaft sich voll hinter eine Sache zu stellen sinkt rapide („*Was ist es denn diesmal?*"). Natürlich können Projekte scheitern, wenn dies jedoch lediglich aus mangelndem Umsetzungswillen passieren kann, stehen die negativen Folgen in keiner Relation zum erwartenden Gewinn.

Anschaffung von „Shelfware"

Schon in den letzten Abschnitten klang immer wieder die Warnung durch, sich nicht durch die Versprechungen von Werkzeugherstellern in die Irre führen zu lassen. Die Flut an Stickern mit Labeln wie *100% MDA kompatibel* auf Produktkartons sowie die Menge an Werbung, die unter dem Stichwort MDA in aktuellen Fachzeitschriften zurzeit existiert, lassen mitunter den Eindruck entstehen, dass die geeignete Kollektion von Tools die meisten Aspekte der MDA schon abdecken wird, also quasi so etwas wie COTS-MDA darstellen [COTS].

Unwidersprochen ist die Infrastruktur eines MDA-Projektes, also die Menge der zur Verfügung stehenden Tools, Metamodelle und Notationen ein Kernpunkt zur Durchführung eines solchen Vorhabens darstellen. Sinnvoll ist unserer Meinung nach ein gesunder Mix aus bestehenden Produkten und Eigenentwicklung. Das Fallbeispiel des Buches zeigt, wie eine solche Mischung aus kommerziellen und eigenentwickelten Infrastrukturbausteinen aussehen kann und welche Komponenten benötigt werden. Der folgende Text soll vermitteln, warum vorsichtige Zurückhaltung beim Kauf von MDA-Produkten geboten ist, die als *Out-of-the-box* Anwendungen angeboten werden:

Unter dem Begriff *Shelfware* wird die Beobachtung des Phänomens gefasst, dass viele Entwickler zwar grundsätzlich großen Spaß am Ausprobieren neuer Produkte wie Entwicklungsumgebungen etc. haben, den anfänglichen Enthusiasmus jedoch schnell wieder verlieren, wenn diese entweder ihre Versprechungen nicht halten oder die alten Tools, mit deren Bedienung sie voll vertraut sind, dieselben Aufgaben erledigen. Zum Ärger der Verantwortlichen wandern teuer angeschaffte Produkte dann schnell in den Schrank bzw. ins Regal (daher der Name) und werden schlicht nicht mehr eingesetzt: Die Investition ist vergeudet. Dies gilt vor allem bei oben erwähnten All-in-One-Tools, die die volle Unterstützung des MDA Lifecycles versprechen.

Vorsicht ist also geboten, wenn beabsichtigt wird ein solches Produkt anzuschaffen. Diese Produkte sind oft nicht nur extrem teuer in der Anschaffung, viele der erwähnten Standards, die später ein reibungsloses Zusammenarbeiten von Tools verschiedener Hersteller ermöglichen sollen, befinden sich zur Zeit der Drucklegung noch in der Finalisierungsphase. Die auf dem Markt befindlichen Werkzeuge sind daher, trotz aller Versprechungen, in hohem Maße proprietär und wecken daher unangenehme Erinnerungen an die Versprechungen von CASE [CASE]. Zudem verbietet eine teure Komplettlösung oftmals die iterative Einführung des neuen Ansatzes in den Projektalltag. Wer nur Teilaspekte ausprobieren will, lässt Funktionalität der angeschafften Werkzeuge ungenutzt und verschenkt damit Geld, weshalb nach dem Kauf oft das Alles-oder-Nichts Prinzip den Vorzug erhält, auch wenn die Struktur des Unternehmens nicht für eine solche Vorgehensweise geeignet ist (vgl. dazu Abschnitt 6.4).

Abschnitt 6.6.2 erläutert das Prinzip der offenen Technologien, das zur Vermeidung genau dieser Problematik zur Anwendung kommen

kann. Das Fallbeispiel kommt sogar ganz ohne die Anschaffung kommerzieller Software aus.

Code Monkeys/Monkey Work

Wie in Abschnitt 6.3 kurz angerissen erfordert die Anwendung eines modellgetriebenen Entwicklungsansatzes Anpassungen am klassischen Rollenmodell. Dadurch, dass ein Großteil des Anwendungsentwurfs bereits durch das Framework vorgegeben wird schränkt sich der Entscheidungsspielraum für die Gruppe der Anwendungsingenieure mehr oder minder stark ein. Für die Anwendungsentwickler (vulgo Programmierer), im zuvor erwähnten Abschnitt Softwareentwickler genannt, trifft dies im besonderen Maße zu, da „ihr" Artefakt, d.h. das für die Anwendungssoftware präskriptive Systemmodell, im MDA-Sinne – so zumindestens die Theorie – fachlich geprägt sowie frei von technischen Aspekten ist und damit kausaler Weise nahe an der Analyse bzw. am Analysemodell liegt. Sind die Lieferungen der Analytiker sehr detailliert bzw. nehmen sie die Implementierung (in Teilen) vorweg, so besteht mehr denn je die Gefahr, dass die Arbeit der Entwickler zu stupider Translation verkommt, sie quasi zur *Monkey Work* entartet und Entwickler somit nur noch als *Code Monkeys* fungieren. Erfolgt die Übergabe der Analyseergebnisse als „Wurf über den Zaun" so verschärft dies die Problematik zusätzlich.

Um sicher zu stellen, dass Analytiker und Entwickler ausreichend Platz zur Entfaltung ihrer Kreativität erhalten, sollte man vor diesem Hintergrund die Aufgaben- bzw. Kompetenzbereiche von Analytikern und Entwicklern vor Projektbeginn dahingehend auf den Prüfstand stellen und ggf. neu abstecken. Werden die Bereiche daraufhin zu klein, so kann die Ausübung der beiden angesprochenen Rollen in Personalunion eine mögliche Altertnative sein.

Schweigen im Architektenturm

Wie bereits in Abschnitt 6.3 über die Rollen und Aufgaben innerhalb eines MDA-Prozesses immer wieder herausgestrichen, ist die effektive Kommunikation eine der Grundvoraussetzungen zum Gelingen des gesamten Unterfangens. Hierzu sind vor allem die Rollen im Domain Engineering sowie die führenden Rollen des Application Engineering (die wir hier einmal als die „Architekten" bezeichnen wollen) unbedingt mit Personen zu besetzen, die nicht nur die Fähigkeit besitzen technische Wunderwerke zu vollbringen, sondern diese auch den anderen Projektbeteiligten mitteilen und erläutern zu können. Wie im Abschnitt über *Agile MDA* (vgl. Abschnitt 6.6.2) angeklungen, ist nicht lediglich die kurzfristige Erledigung eines Projektes durch ein oder zwei Einzelkämpfer das Ziel, sondern die

langfristige Wissenssicherung und Multiplikation dieses Wissens im gesamten Unternehmen.

Auch die als kritisch befundene Aktivität des *Feedbacks* (Abschnitt 6.2.6) zur Verbesserung und Evolution des gesamten Ansatzes kann nur funktionieren, wenn die Architekten- und Expertenrollen so besetzt sind, dass Einwände und Erfahrungen der Entwickler auch in die Aktivitäten des Domain Engineerings einfließen und *nicht* die Elfenbeinturm-Mentalität einreißt. Also: Technik ist nicht alles, Finger weg von der Besetzung tragender Entwicklungsrollen mit brillianten Technikern, die keine kommunikativen Fähigkeiten besitzen.

Not-invented-here Syndrom

Erliegen Sie nicht der Versuchung, plötzlich alles selbst erfinden zu wollen, nur weil der *Transformationshebel* und die damit verbundenen Produktivitätszuwächse Ihnen die Möglichkeit dazu suggerieren. So sollte man beispielsweise nicht auf die ambitionierte Idee plötzlich unbedingt sein eigenes Persistenz-Framework, den eigenen Transaktionsmonitor, etc. entwickeln zu müssen. Nutzen Sie stattdessen bestehende Frameworks bzw. Plattform-Stacks: Zum einen sind diese erprobt und haben sich bereits in prä-MDA-Zeiten in der Praxis bewährt, zum anderen wird damit die transformatorisch zu überbrückende Abstraktionslücke klein gehalten und folglich die Transformationsregeln vereinfacht, was den Aufwand zur Pflege und Implementierung reduziert (vgl. Kapitel 5).

Standard-Hörigkeit

Die Bestrebungen der OMG im Umfeld der MDA manifestieren sich nach und nach in einem Fundus verschiedener Standards. Diese Standards sind sicherlich eine der Stärken des Ansatzes, vermögen sie doch die potenzielle Gefahr eines Vendor-Lock-in, der als einer der Hauptgründe für das Scheitern der CASE-Strömung erachtet wird (vgl. obige Ausführungen zur Anschaffung von Shelfware), zu vermeiden.

Standards sind allerdings mitnichten ein Allheilmittel und bieten oft gerade mal den – meistens „politisch“ motivierten – kleinsten gemeinsamen Nenner der am Standardisierungsprozess beteiligten Parteien. Darüber hinaus adressieren viele Standards zwar ein Praxisproblem, sind aber auf dem Reißbrett entworfen worden ohne erprobte Praxislösungen einzubeziehen. Beispielsweise wird die durch den Java Community Process (JCP) standardisierte Spezifikation für Objekt-Persistenz mit dem Namen Java Data Objects (JDO) [JSR12] diesbezüglich immer wieder als *Dead-before-born* gehandelt. Das Rennen im Bereich der Persistenz-Frameworks, um beim

Beispiel zu bleiben, haben hingegen Praxislösungen wie Hibernate gemacht (siehe Praxisteil dieses Buches). Wie es das Schicksal will halten dessen „Best Practices“ wiederum Einzug in die nächste Generation des EJB-Standards a.k.a. EJB 3.

Bei den OMG-Standards bietet sich mitunter ein durchwachsenes Bild. Die MOF hat beispielsweise erst mit der Version 2.0 die Praxisinfusion erhalten, die sie tauglich für den Alltagseinsatz werden lässt (vgl. Erörterungen zum Thema *Eclipse Modeling Framework* in Kapitel 8). Ganz ähnlich ergeht es der UML, die die Best-Of-Breed der Ansätze der drei Amigos bildet. Zu Dokumentationszwecken bereits etabliert, muss sie den Beweis der Praxistauglichkeit für den Einsatz innerhalb modellgetriebener Entwicklung erst noch erbringen, mit der gründlich renovierten Version 2.0 will man genau das tun.

Darüber hinaus decken die derzeit vorliegenden Standards nur einen Teil der Anforderungen ab, die ein modellgetriebener Ansatz mit sich bringt. Im Bereich der Transformationsdefinition wird mit *Query / View / Transformations (QVT)* gerade eine dieser Lücken geschlossen. Für die Domäne der Oberflächenmodellierung hingegen bleibt abzuwarten, ob es dazu zukünftig überhaupt irgendwelche Aktivitäten geben wird – hier ist Eigeninitiative gefragt.

Also … Zusammenfassend lässt sich also das Resümee ziehen darauf zu auchten nicht in devote Standard-Hörigkeit zu verfallen. Im Einzelnen heißt das:

- sich bei der Lösungsfindung nicht durch die verfügbaren Standards künstlich beschränken zu lassen,
- generell vorsichtig gegenüber Standards zu sein und diese einer strengen Betrachtung zu unterziehen. Dies gilt besonders bei Versionen mit der Eitkettierung 1.0 und
- nicht (zulange) auf angekündigte Standards zu warten. Standardisierungsprozesse können sich in die Länge ziehen.

Phlegmatischer Build-Prozess

Bei modellgetriebener Entwicklung sind aufgrund derzeit noch mangelnder Verzahnung in der Tool-Kette, insbesondere hervorgerufen durch die zwangsläufige Koexistenz von Modell und Code, oftmals eine Fülle von Einzelschritten innerhalb eines Entwicklungszyklus zu gehen. Die folgende Liste veranschaulicht das:

- Änderungen im PIM vornehmen

- PIM exportieren
- PIM-zu-PSM Tranformation durchführen
- Änderungen im PSM vornehmen
- PSM exportieren
- Code generieren
- Änderungen im Code vornehmen
- Code compilieren
- Executables paketieren
- Executables deployen
- Tests durchführen
- etc.

Zugegebenermaßen war/ist eine Vielzahl der angegebenen Schritte auch in nicht-modellgetriebenen Ansätzen notwendig. Die zusätzlich erforderlich gewordenen Schritte blähen den Zyklusdurchlauf jedoch abermals auf. Je nach Art und Umfang des Projektes kann die Entwicklungsarbeit dadurch so schwerfällig werden, dass dies die Produktivitätsgewinne eines modellgetriebenen Vorgehens schnell wieder zunichte macht.

Sorgen Sie also dafür, dass der Build-Prozess weitestgehend ohne händische Eingriffe ablaufen kann (One-Step Build-Prozess), sich aus Modell-/Code-Editor starten lässt und sich wahlweise ganz oder auch nur teilweise durchführen lässt.

Literatur

[AM] Kent Beck et al.: *Manifesto for Agile Software Development*.
http://agilemanifesto.org/ (letzter Abruf Mai 2005)

[BKPS04] Günter Böckle, Peter Knauber, Klaus Pohl und Klaus Schmid (Hrsg.): *Software-Produktlinien – Methoden, Einführung und Praxis*. dpunkt, 2004.

[Boe95] Barry W. Boehm: *Anchoring the Software Process*. USC CSE Technical Reports, 1995.
http://sunset.usc.edu/publications/TECHRPTS/1995/usccse95-507/ASP.pdf (letzter Abruf Mai 2005)

[Bos01] Jan Bosch: *Software Product Lines – Organizational Alternatives*. In: Proceedings of the 23rd International Conference on Software Engineering (ICSE'01), S. 91-100, IEEE Computer Society, 2001.

[CASE] Wikipedia: *Computer-aided software engineering*.
http://en.wikipedia.org/wiki/CASE (letzter Abruf Mai 2005)

[CC] ThoughtWorks Inc.: *CruiseControl Home*.
http://cruisecontrol.sourceforge.net/ (letzter Abruf Mai 2005)

[CF] Wikipedia: *Creeping featurism.*
http://en.wikipedia.org/wiki/Creeping_featurism (letzter Abruf Mai 2005)

[CI] Martin Fowler und Matthew Foemmel: *Continuous Integration.*
http://www.martinfowler.com/articles/continuousIntegration.html (letzter Abruf Mai 2005)

[CN01] Paul Clemens und Linda Northrop: *Software Product Lines – Practices and Patterns*. Addison-Wesley, 2001.

[Coc01] Alistair Cockburn: *Agile Software Development*. Addison-Wesley, 2001.

[Coc04] Alistair Cockburn: *Crystal Clear – A Human-Powered Methodology for Small Teams*. Addison-Wesley, 2004.

[Col95] Andy Cole: *Runaway Projects – Cause and Effects*. In: Software World, Volume 26, Number 3, S. 3-5, A. P. Publications, 1995.

[COTS] Wikipedia: *Commercial off-the-shelf*. http://en.wikipedia.org/wiki/COTS (letzter Abruf Mai 2005)

[DeM01] Tom DeMarco: *Slack – Getting Past Burnout, Busywork, and the Myth of Total Efficiency*. Broadway Books, 2001.

[DL99] Tom DeMarco und Timothy Lister: *Peopleware – Productive Projects and Teams, Second Edition*. Dorset House, 1999.

[Eclipse] Eclipse.org: *Eclipse.org Main Page*. http://www.eclipse.org/ (letzter Abruf Mai 2005)

[Gla99] Robert L. Glass: *The Realities of Software Technology Payoffs*. In: Communications of the ACM, Volume 42, Issue 2, S. 74-79, ACM, 1999.

[Jakarta] Apache Software Foundation (ASF): *The Apache Jakarta Project*. http://jakarta.apache.org/ (letzter Abruf Mai 2005)

[JBoss] JBoss Inc.: *JBoss.com – The Professional Open Source Company*. http://www.jboss.org/index.html (letzter Abruf Mai 2005)

[JSR12] The Java Community Process: *JSRs – Java Specification Requests – detail JSR# 12*. http://www.jcp.org/en/jsr/detail?id=12 (letzter Abruf Mai 2005)

[LM97] Wing Lam und John A. McDermid: *A summary of domain analysis experience by way of heuristics*. In: Proceedings of the 1997 symposium on Software reusability, S. 54-64, ACM, 1997.

[OWS+03] Bernd Oesterreich, Christian Weiss, Claudia Schröder, Tim Weilkiens und Alexander Lenhard: *Objektorientierte Geschäftsprozessmodellierung mit der UML*. dpunkt, 2003.

[POC] Wikipedia: *Proof of concept*. http://en.wikipedia.org/wiki/Proof_of_concept (letzter Abruf Mai 2005)

[Se04] Sekørb Defla: *Innovation considered harmful*. In: Proceedings of the 2004 symposium on Innovative Software Engineering, S. 54-64, AB, 2004.

[SPEM] Object Management Group (OMG): *Sofware Process Engineering Metamodel, Version 1.1*. http://www.omg.org/technology/documents/formal/spem.htm (letzter Abruf Mai 2005)

[SPL03] Robert C. Seacord, Daniel Plakosh und Grace A. Lewis: *Modernizing Legacy Systems – Software Technologies, Engineering Processes, and Business Practices*. Addison Wesley Professional, 2003.

[TPTP] Eclipse.org: *Eclipse Test & Performance Tools Platform*. http://www.eclipse.org/tptp/index.html (letzter Abruf September 2005)

[U2TP] The U2TP Consortium: *U2TP – The UML Testing Profile*. http://www.fokus.gmd.de/u2tp/ (letzter Abruf September 2005)

[UMLProfiles] Object Management Group (OMG): *MDA Specifications – UML Profiles*. http://www.omg.org/mda/specs.htm#Profiles (letzter Abruf Mai 2005)

[Ver02] Gerhard Versteegen (Hrsg.): *Software-Management*. Springer, 2002.

[WeLa99] David M. Weiss und Chi Tau Robert Lai: *Software Product-Line Engineering – A Family-Based Software Development Process*. Addison-Wesley, 1999.

7 Vorstellung des Fallbeispiels

„Do, or do not. There is no try."

Yoda

In den folgenden Kapiteln wollen wir Ihnen ein Fallbeispiel vorstellen, an dem wir die in den vorangegangen Teilen des Buches doch eher theoretischen Ausführungen zur Anwendung bringen. Gegenstand unserer Betrachtung soll dabei ein fiktives Unternehmen namens *M&M* sein. Die Beschreibung erfolgt dabei durch die Angabe und Erläuterung von Ausschnitten eines ebenso fiktiven Projektverlaufs, das mittels der bisher vorgestellten MDA-Prinzipien durchgeführt wird.

Dieses Vorgehen soll es ermöglichen:

- die Ausgangssituationen zu beschreiben, die es sinnvoll erscheinen lassen, die Anwendung der MDA in Betracht zu ziehen
- ein Gefühl dafür zu bekommen, wie sich die MDA innerhalb eines Projektverlaufs einfügt
- sowie die entwickelten Techniken und angewandten Methoden in einem realistischen Rahmen zum Einsatz kommen zu lassen.

In den folgenden Kapiteln erfolgt schließlich die ausführliche Beschreibung des Projektes, das die benötigte Software und damit eine Art *Proof-of-Concept* für den MDA-Ansatz liefern soll. Wir wollen dabei davon ausgehen, dass die Durchführung eines solchen MDA-Projektes für uns völliges Neuland ist, wir also auch die meisten Tools, die gebraucht werden, erst selbst planen und erstellen müssen, bevor wir an das „eigentliche" Problem denken können.

Gerade für solche ersten Schritte, die womöglich eine Änderung in der unternehmensweiten Vorgehensweise zur Softwareentwicklung einläuten, bietet sich die Durchführung als *Pilotprojekt* an (vgl. auch Kapitel 6). Unser Fallbeispiel, und das damit verbundene Software-Entwicklungsprojekt, haben wir dahingehend ausgerichtet. Es weist dementsprechend die nötige Ernsthaftigkeit auf, die umzusetzende Funktionalität ist aber nicht hochkritisch (siehe unten).

Tun wir einfach so als ob …

Obwohl das beschriebene Projekt kein reales Projekt darstellt, sondern vielmehr eine „Normalisierung" unserer Erfahrungen aus vielen Einzelprojekten abbildet, wollen wir im Folgenden so tun, als würden wir mitten in ein laufendes Projekt hineinschauen, um Ausdrücke wie „Nun wäre…", „So sei … gewesen …", etc. der besseren Lesbarkeit halber zu vermeiden.

Beginnen wollen wir unsere Betrachtung in diesem Kapitel mit der Vorstellung der fachlichen Domäne unseres Projektes, die wir ganz im Sinne der MDA in einem Computation Independent Model (CIM) festhalten. Aus Kapitel 4 wissen wir bereits, dass dieses Modell so etwas wie den Einstieg in den Lebenszyklus der MDA Modelle bildet, indem es zunächst den Kontext des zu entwickelnden Systems fokussiert.

Bevor wir jedoch soweit ist, lassen Sie uns zuvor kurz die Ausgangssituation unseres fiktiven Unternehmens beleuchten.

7.1 Ausgangssituation

Das Unternehmen …

M&M ist ein so genannter Versicherungsmakler, der Kunden in Versichrungsfragen berät und ihnen entsprechende Produkte vermittelt. Für diese Vermittlung erhält das Unternehmen von den eigentlichen Produktgebern – den Versicherungsunternehmen – eine Provision, die auch Courtage genannt wird. Neben diesen Provisionseinnahmen stützt sich das Geschäftsmodell auf Einnahmen aus dem Handel von Adressdaten. Dazu sammelt das Unternehmen so genannte weiche Akquisedaten (also z. B. Hobbies, Wohngegend, Lebenssituation und Umfeld ihrer Kunden).

… und seine Probleme

Unser Unternehmen sieht sich zunehmend mit einer Reihe von Problemen konfrontiert:

Ineffiziente Prozesse

- Die Geschäftsprozesse des Unternehmens sind zwar effektiv, lassen aber an Effizienz mangeln. Sie sind über einen längeren Zeitraum „gewachsen", wurden jedoch niemals explizit definiert und dokumentiert. Vielmehr „weiß jeder im Unternehmen", so

in etwa, „was er zu tun hat“. Hierdurch leidet die Kommunikation der einzelnen Organisationseinheiten untereinander und die Einarbeitung neuer Mitarbeiter wird somit schwieriger und langwieriger. Durch den jüngsten Wachstumsschub des Unternehmens tritt dieser Effekt verstärkt zu Tage.

- Die DV-Landschaft des Unternehmens zeichnet sich durch eine ausgeprägte Heterogenität und einen geringen Automatisierungsgrad aus. Die einzelnen Anwendungssysteme sind kaum bis gar nicht miteinander integriert. Die anfallenden Daten müssen häufig redundant erfasst werden, beispielsweise beim Übergang von Bedarfsanalyse zu Angebot respektive Angebot zu Antrag bzw. Vertrag. Diese Mehrfacherfassung ist fehlerträchtig und kostspielig. Die ständigen Medienbrüche innerhalb der einzelnen Vorgänge erschweren die Arbeit zusätzlich.

Fehlende Anwendungsintegration

- Die Möglichkeit sich über das Internet darzustellen oder gar Geschäfte darüber abzuwickeln hat das Unternehmen bislang gänzlich ignoriert. Es fehlt jegliche E-Business Strategie. Die Konkurrenz hingegen präsentiert sich werbewirksam im Internet und bietet einen Teil ihres Dienstleistungsportfolios dort an. Ein Teil ehemaliger *M&M* Kundschaft nutzt aus diesem Grund bereits die Dienstleistungsangebote der Konkurrenz.

Fehlende E-Business Strategie

- In jüngster Zeit klagen die Mitarbeiter des Call-Centers über zunehmende Überlastung. Oftmals sind die Leitungen durch Gespräche belegt, in denen Kunden beispielsweise die Änderungen ihrer Anschrift oder Bankverbindungsdaten bekannt machen möchten. So gehen Beratungs- und Vermittlungskapazitäten verloren. Zu gewissen Stoßzeiten, bedingt etwa durch anstehende Gesetzesänderungen zum Jahreswechsel, konnten so beispielsweise eine Reihe von Anfragen nicht entgegengenommen bzw. bearbeitet werden.

Fehlende Ressourcen

- […]

Dem Unternehmen selbst ist seine problematische Lage nach und nach bewusst geworden. Durch den zunehmenden Wettbewerbsdruck ist man schließlich zum Ergebnis gelangt, verstärkt nach Optimierungsmöglichkeiten suchen zu müssen. Da in diesem Bereich allerdings kein eigenes Know-how vorhanden ist, hat man sich auf Vorstandsebene dazu entschlossen externe Hilfe in Anspruch zu nehmen. Die Geschäftsleitung des Unternehmens ist daher mit der Anfrage an uns herangetreten, eine Strategie zu entwickeln, die den beschriebenen Problemen entgegengesetzt werden kann. Aufgrund der groben Problemstruktur ist die Vorgabe, dabei gezielt nach

Möglichkeiten zu suchen, die Geschäftsprozesse vermehrt IT-gestützt und über das Internet zu betreiben.

Geplantes Vorgehen

Unser Vorgehen haben wir dementsprechend in zwei Bereiche untergliedert:

- die *Modellierung und Analyse der Geschäftsprozesse*, inklusive der Identifizierung der Prozesse, die sich für eine (Teil-) Automatisierung eignen
- und der *Erstellung der dazu benötigten Software und Methodik*, von der Anforderungsanalyse über den Entwurf hin zur Implementierung, mittels eines MDA-basierten Ansatzes.

OOGPM

Unser Vorgehen zur Geschäftsprozessmodellierung orientiert sich an der OOGPM (Objektorientierte Geschäftsprozessmodellierung), die in [OWS+03] und in Anhang B dieses Buches genauer beschrieben ist. Der Darstellbarkeit wegen haben wir im beschriebenen Szenario die Geschäftsprozessmodellierung fallabschließend und getrennt von der Systementwicklung durchgeführt. Prinzipiell hätte sie sich aber auch in die iterativ-inkrementelle Vorgehensweise unserer Systementwicklung integrieren lassen, um so das Modell des Geschäftssystems wie das Systemmodell sukzessive zu entwickeln (vgl. Kapitel 6 – Iterativ-Inkrementelle Softwareentwicklung).

7.2 Modell des Geschäftssystems

Beim Reengineering von Geschäftsprozessen sollte zunächst die Modellierung des Ist-Zustands der aktuell greifenden Geschäftsprozesse in Angriff genommen werden. Auch wenn der Name der Disziplin vielleicht zunächst anderes vermuten lässt, wird nicht nur die Organisation der Prozesse und des Ablaufs, sondern auch die des Aufbaus des betrachteten Unternehmens festgehalten.

Ergebnis dieser Bemühungen sind eine Reihe von Modellen bzw. Diagrammen, die unter die bereits bekannte Kategorie der CIMs fallen. Sie sind unabhängig von einer möglichen systemtechnischen Umsetzung und nahe an der betriebswirtschaftlichen Sicht situiert.

Der Gebrauch dieser Ergebnisse beschränkt sich nicht auf die Disziplin des Prozess-Engineerings, sondern zieht sich durch das gesamte Projekt. Die nächste Phase nach den Prozessbetrachtungen wird so die Ableitung der Systemanforderungen aus den so erarbeiteten Modellen sein.

Nach diesen vorbereitenden Bemerkungen, wollen wir nun mit der Modellierung der Prozesse beginnen.

Los gehts

7.2.1 Modellierungsfokus

Zunächst wird der Modellierungsfokus festgelegt. Dieses Vorgehen hat den Zweck, sich gezielt auf den, für uns relevanten, Ausschnitt des analysierten Unternehmens konzentrieren können. Bei größeren Unternehmen kann dies allein der schieren Komplexität wegen unerlässlich sein.

Abb. 7.1 Modellierungs-fokus

	GPM-Betrachtungsgegenstand festlegen
Modellierungsfokus	In der Geschäftsprozessmodellierung wird das Unternehmen *M&M* betrachtet: Organisationseinheiten ▪ Produkte ▪ Vertrieb & Betreuung ▪ Controlling und Rechnungswesen Schwerpunkte (*muss*) ▪ Anbahnung / Akquisition ▪ Bedarfsanalyse ▪ Angebotserstellung / Antrag / Abschluss Zudem sollte betrachtet werden (*kann*) ▪ Provisionsabrechnung Nicht zu betrachten sind ▪ Marketing, Adressdatenhandel
Leitbild (Mission)	„Unabhängige Versicherungsberatung und -vermittlung für unabhängige Menschen“
Leitziel (Vision) und Kennzahlen	Die Zahl der Bestandskunden in 2 Jahren um 50 % erhöhen Kennzahl: Anzahl der Bestandskunden
Unternehmenszweck	Geld verdienen durch Beratung und Vermittlung von Versicherungsprodukten

Aber auch ein kleineres Unternehmen vollständig abbilden zu wollen kann zum hoffnungslosen Unterfangen verkommen. Die Fokussierung verhindert, dass wir uns an irrelevanten Stellen unnötigerweise lang aufhalten und uns dadurch wohlmöglich die Zeit fehlt uns in der gebotenen Weise den Kernkonzepten zu widmen.

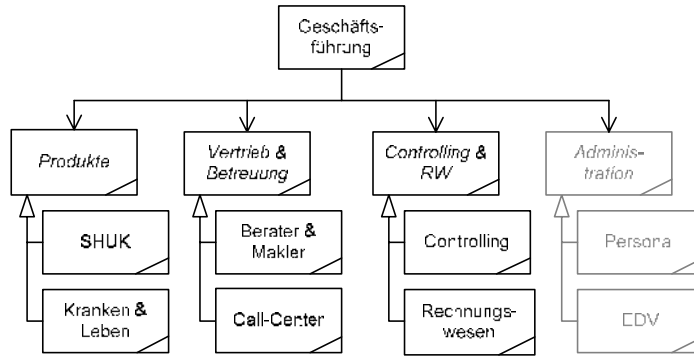
An diesem Punkt wird dabei nicht nur festgehalten *was*, sondern auch explizit abgegrenzt, *was nicht* betrachtet werden soll. Diese Auswahl lässt sich an Organisationseinheiten oder Geschäftsprozessen festmachen. Initial werden dabei auch bereits Schwerpunkte gelegt und entsprechend in *muss* und in *kann* klassifiziert.

Wie aus Abb. 7.1 zu entnehmen ist, schließen wir gezielt die Organisationseinheit `Marketing` sowie den Bereich des `Adressdatenhandels` aus der vorläufigen Betrachtung aus. Diese Selektion gründet sich richtet sich nach der Dringlichkeit der einzelnen Maßnahmen (vgl. Ausgangssituation). Schwerpunktmäßig wollen wir uns daher eher mit den unmittelbar operativen Bereichen, also mit Akquisition/Anbahnung, Bedarfsanalyse, Angebotserstellung, Antrag, Abschluss etc., befassen.

7.2.2 Organisationseinheiten

Um eine bessere Übersicht zu erhalten, welche Unternehmensbereiche Aufgaben im Rahmen des zuvor festgelegten Modellierungsfokus wahrnehmen, bzw. welche Beziehungen und Verantwortlichkeiten zwischen diesen Bereichen bestehen, ist es hilfreich die Organisationsstruktur des Unternehmens mittels eines Organigramms festzuhalten.

Abb. 7.2 Darstellung der Organisationseinheiten im Modell



In Abb. 7.2 sehen wir die modellierte Organisationsstruktur von *M&M*. Auch an dieser Stelle wird zur Modellierung das UML-Profil der OOGPM verwendet. Rechtecke stellen Organisationseinheiten (OE) dar. Durchgezogene Pfeile zeigen die *disziplinarische Weisungsbefugnis* in Richtung Pfeilspitze an. Die Generalisierung behält ihre übliche Bedeutung.

Bei der Festlegung des Modellierungsfokusses im vorigen Abschnitt wurde der Bereich `Administration` weder ein noch ausgeschlossen, da zum Zeitpunkt der Modellerstellung nur unzureichendes Wissen über die vorhandenen Abteilungen vorlag. Bei der Erstellung des Organigramms wird dieser nun explizit ausgeschlossen (ausgegraute Darstellung).

Weiteres verwandtes Notationselement: Die drei Organisationseinheiten `Produkte`, `Vertrieb & Betreuung` respektive `Controlling & RW`, die wir bereits zuvor in unseren Modellierungsfokus aufgenommen hatten, sind als abstrakte Einheiten dargestellt (kursiver Schriftsatz). Abstrakt heißt in diesem Zusammenhang, dass etwa die OE `Produkte` eine Zusammenfassung der konkreten Einheiten `SHUK (Sach-, Haftpflicht-, Unfall- und Kraftfahrt-Versicherungen)` und `Kranken & Leben` ist, sie als solche aber z. B. keine ausgewiesene Leitung hat.

7.2.3 Geschäftspartner

Im nächsten Schritt werden die *Geschäftspartner (kurz: GP)* des betrachteten Unternehmens identifiziert. Man unterscheidet vor allem zwischen *aktiven* und *passiven* GP.

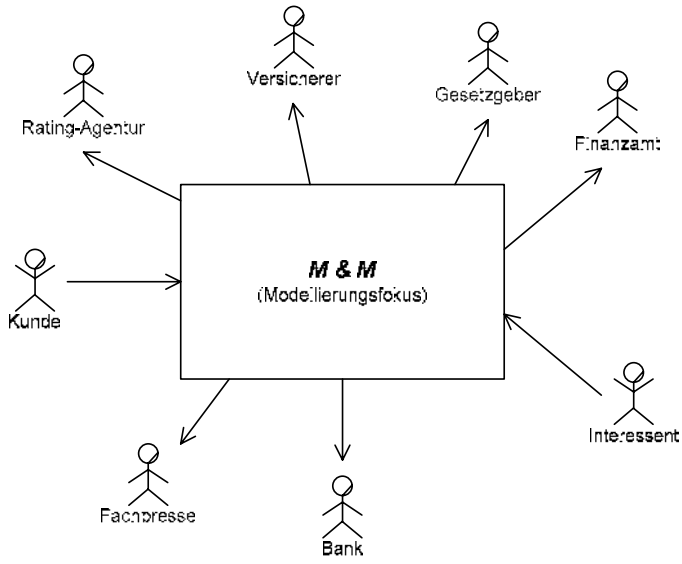


Abb. 7.3 Geschäftspartner

Gemeinsam ist beiden GP-Klassen, dass sie außerhalb des Unternehmensbereiches stehen, und mit dem Unternehmen in irgendeiner

Weise in Kontakt treten. Aktive Geschäftspartner initiieren allerdings selbstständig Geschäftsvorgänge und stellen in dem meisten Fällen die Kunden eines Unternehmens dar; Passive Geschäftspartner werden lediglich durch eine vorangehende Initiative des betrachteten Unternehmens in die Geschäftsprozesse eingebunden (z. B. Ausgabe einer Mitteilung an die Fachpresse). Abb. 7.3 zeigt die Geschäftspartner von *M&M*.

Auch ohne große Mühe wurden als erstes die aktiven GP `Kunde` und `Interessent` gefunden. Im Workshop, der diesen Findungsprozess begeleitet hat, und an dem Vertreter aller Organisationseinheiten beteiligt waren, wurden im Brainstorming und dem Ansatz geeigneter Elicitation-Techniken (vgl. etwa [CK92]), nach und nach auch die wichtigsten passiven GP ermittelt, etwa `Versicherer` und `Banken`, aber auch Institutionen wie `Gesetzgeber` und `Finanzamt` mit denen *M&M* weniger häufig in Kontakt tritt.

Obwohl an dieser Stelle noch nicht im Fokus des Interesses, wurden bereits hier *Geschäftsmitarbeiter* sowie erste *Geschäftsanwendungsfälle* gefunden. Diese wurden notiert und dann an späterer Stelle in die Modellierung aufgenommen (siehe folgenden Abschnitt).

7.2.4 Geschäftsanwendungsfälle der aktiven Geschäftspartner

Ausgehend von den aktiven Geschäftspartnern beginnen wir nun die *Geschäftsanwendungsfälle (GAF)* zu identifizieren. Wie beschrieben passiert es manchmal, dass Geschäftsanwendungsfälle schon bei der Ermittlung der Geschäftspartner genannt werden. Ist absehbar, dass die meisten GAF schon in dieser Phase ermittelt werden können, können die beiden Arbeitsschritte auch zusammengelegt werden. Abb. 7.4 zeigt einen Ausschnitt aus den gefundenen GAF der aktiven Geschäftspartner.

Man unterscheidet zwischen *Kern-* und *Unterstützenden Geschäftsanwendungsfällen*. Diese Unterscheidung gerät recht einfach: Kern-GAF sind solche, die von aktiven GP angestoßen werden *und* direkt mit den Zielen des Unternehmens Gewinn zu erwirtschaften zusammenhängen. GAF, die in diese Kategorie fallen werden mit zwei Schrägstrichen gekennzeichnet. Alle anderen fallen in die zweite Kategorie und erhalten nur einen Strich.

Detaillierung wichtiger GAF

Bis zu diesem Zeitpunkt haben wir uns lediglich einen kurzen Überblick über die Gesamtheit aller GAF verschafft und die Detaillierung einem späteren Zeitpunkt vorbehalten. Im Folgenden wollen

wir uns den Geschäftsanwendungsfall `Angebot erstellen` herausgreifen, der in einem späteren Kapitel von besonderem Interesse für uns sein wird.

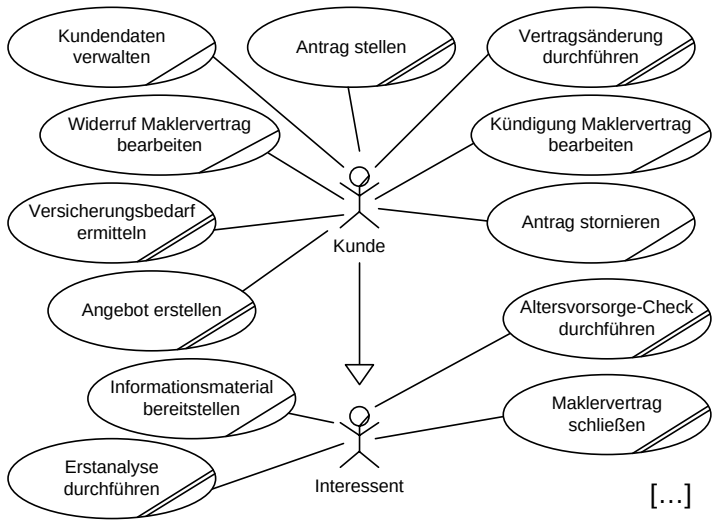


Abb. 7.4 Geschäftsanwendungsfälle der aktiven Geschäftspartner

Ein Kunde hat die Möglichkeit sich von *M&M* ein Angebot für den Neuabschluss oder aber eine Änderung eines Versicherungsvertrages erstellen zu lassen.

	Beschreibung Geschäftsanwendungsfall
Name	Angebot erstellen
Art	Kern-Geschäftsanwendungsfall
Kurzbeschreibung	Ein Kunde lässt sich ein Angebot für eine Versicherung von *M&M* erstellen.
Auslöser, ggf. Motivation	Ein Kunde wendet sich an *M&M*, um ein Angebot für einen Neuabschluss oder eine Änderung eines Versicherungsvertrages zu erhalten.
Ergebnis	Für den Kunden wurde ein Angebot für eine Versicherung erstellt.
Akteure	Kunde

Abb. 7.5 Kurzbeschreibung des GAF Angebot erstellen

Neben der Darstellung von konkreten Informationen eines ausgewählten Produktes berücksichtigt ein solches Angebot üblicherweise auch die Angabe möglicher Produktalternativen.

Abb. 7.5 zeigt die Kurzbeschreibung dieses Geschäftsanwendungsfalls. Sagt das Angebot dem Kunden zu, so kann er über *M&M* einen entsprechenden `Antrag stellen` (lassen), der dann an das entsprechende Versicherungsunternehmen weitergeleitet wird.

7.2.5 Weitere unterstützende Geschäftsanwendungsfälle

Abb. 7.6 Weitere Geschäftsanwendungsfälle

Als nächster Schritt erfolgt die Ermittlung weiterer unterstützender Geschäftsanwendungsfälle, die eventuell im ersten Durchgang „vergessen“ wurden. Abb. 7.6 zeigt einen Ausschnitt der so ermittelten Anwendungsfälle.

Unterstützende GAF

Unterstützende Geschäftsanwendungsfälle sind Vorgänge, die von Mitarbeitern zur Erfüllung dieses Kundenwunsches oder schlicht zum Funktionieren des Gesamtbetriebes beitragen. Beispielhaft sei hier der GAF `Analysevorlage pflegen` herausgegriffen, der für sich genommen keinen geschäftlichen Mehrwert bringt, für Kern-Geschäftsanwendungsfälle wie `Erstanalyse durchführen` oder `Versicherungsbedarf ermitteln` aber die notwendige Basis in Form von Analysevorlagen zur Verfügung stellt.

7.2.6 Geschäftsmitarbeiter/Akteurmodelle

Von den gefundenen Geschäftsanwendungsfällen ausgehend, werden nun die Akteure innerhalb des Unternehmens – sprich die *Mitarbeiterrollen (MA-Rollen)* identifiziert.

Innen- und Außenorientierte MA

Unterschieden wird zwischen *innen- und außenorientierten MA*. Außenorientierte MA sind solche, die direkten Kontakt mit den Geschäftspartnern des Unternehmens unterhalten, während innenorientierte MA sich im Wesentlichen auf interne Tätigkeiten und den

Kontakt zu weiteren MA beschränken. Gleichzeitig wollen wir Einzelheiten über die Beziehungen festhalten, die die ermittelten Mitarbeiter zu bereits gefundenen Elementen der bis zu diesem Zeitpunkt durchgeführten Geschäftsprozess-Analyse (GPA) besitzen.
Bei diesem „In-Bezug-setzen“ der bisherigen GPA-Ergebnisse spricht man von der Modellierung des *Kontextes* der Mitarbeiter. Abb. 7.7 zeigt ein Resultat dieser Bemühungen.

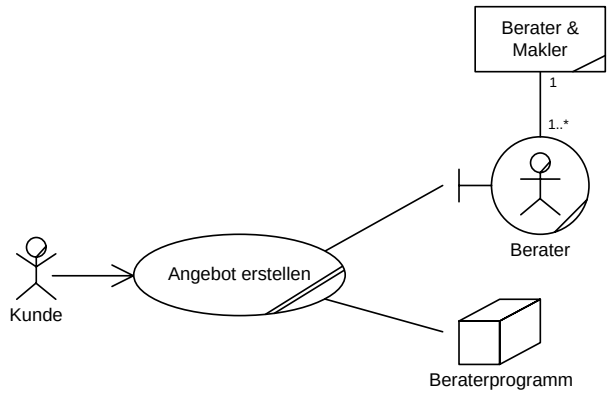


Abb. 7.7 Geschäftsmitarbeiter im Kontext Angebot erstellen

Wir beschränken uns auch hier auf den GAF `Angebot erstellen`: `Kunden` können den GAF anstoßen, die Erstellung des Angebotes wird dann von einem zuständigen `Berater` unter Benutzung einer Software, die hier `Beraterprogramm` genannt wird, durchgeführt. Über Zeit und Ort der Handlung sagt das Diagramm nichts aus, die Anfertigung des Angebots kann sowohl in einer der Geschäftsstellen von *M&M* als auch Vorort beim Kunden erfolgen.

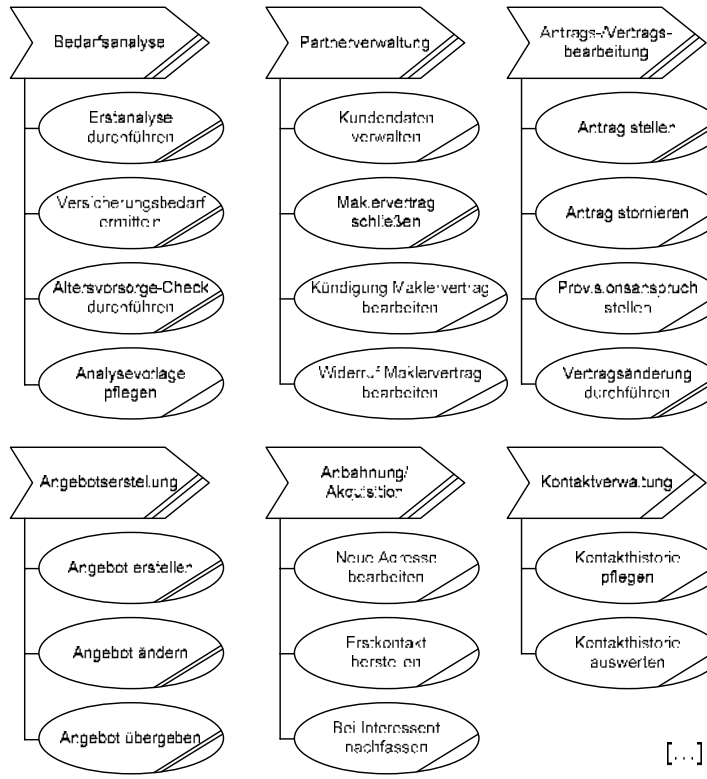
Der Kontext verrät auch, dass `Berater` der Organisationseinheit `Berater & Makler` zughörig sind und es sich bei dieser Gruppe um außenorientierte MA handelt, was durch das verwendete Schnittstellensymbol aus dem Standardrepertoire der UML-Superstructure dargestellt wird.

7.2.7 Geschäftsprozesse

Nachdem wir nun eine Kurzbeschreibung aller wesentlichen Anwendungsfälle im Modellierungsfokus vorliegen haben, können wir nun durch die Gruppierung dieser in thematisch zusammenhängende Einheiten die Modellierung der Geschäftsprozesse des Unternehmens vornehmen. Dies wurde durch die Beschriftung von Karten

mit den Namen der gefundenen GAF und der Durchführung eines weiteren Workshops bewerkstelligt. Zuerst wurden an einer Pinwand thematische Cluster gebildet. Nachdem die Zuordnung einigermaßen stabil zu sein schien, versucht Namen für die gefundenen Gruppen zu finden. Das Resultat – die Geschäftsprozesse von *M&M* – ist ausschnittsweise in Abb. 7.8 dargestellt.

Abb. 7.8 Geschäftsprozesszuordnung



Kategorisierung der GP

Auch hier findet – analog zu den Geschäftsanwendungsfällen – wieder eine Unterteilung in *Kern-* und *unterstützende Prozesse* statt. Auch hier existiert eine einfache Faustregel: Enthält der Geschäftsprozess überwiegend Kern-GAF, handelt es sich folgerichtig um einen *Kern-Geschäftsprozess*.

Die Beschreibung der Geschäftsprozesse wurde hier, wie auch zuvor bei den Anwendungsfällen, hybrid auf grafische und textuelle Weise modelliert. Abb. 7.9 zeigt die textuelle Beschreibung des Geschäftsprozesses `Angebotserstellung`. Auch hier wäre wieder die zusätzliche Modellierung des Kontextes der Prozesse möglich (vgl.

Abschnitt 7.2.6), also die Darstellung der gegenseitigen Beziehungen aller bisher gefundenen Analyse-Elemente mit den Hilfsmitteln der UML (Assoziationen, Relationen, …).

	Beschreibung Geschäftsprozess
Name	Angebotserstellung
Art	Kern-Geschäftsprozess
Kurzbeschreibung	Dieser Geschäftsprozess beschreibt alle Sachverhalte, die in direktem Zusammenhang mit der Erstellung von Angeboten stehen.
Enthaltene GAF	▪ Angebotsanforderung entgegennehmen ▪ Angebot erstellen ▪ Angebot ändern ▪ Angebot übergeben ▪ Angebotsüberprüfung abwarten ▪ Rückfrage zu Angebot beantworten ▪ Bei Angebotsanforderndem nachfassen
Verantwortlich	Leitung Berater & Makler
Beteiligt	Geschäftspartner: Kunde Geschäftsmitarbeiter: Berater

Abb. 7.9 Beschreibung des Geschäftsprozesses Bedarfsanalyse

Da die Darstellung an dieser Stelle jedoch keinen weiteren Informationsgehalt für die Weiterführung des Fallbeispiels beinhaltet, wollen wir auf die Präsentation dieser Kontexte verzichten und sofort weiter mit der Analyse fortfahren.

7.2.8 Essenzbeschreibungen der Geschäftsanwendungsfälle

Bisher wurden die Geschäftsanwendungsfälle auf recht kurze Weise beschrieben und dienten lediglich als Überblick und Grundlage zur Ermittlung der korrespondierenden Geschäftsprozesse. Da dieser Detaillierungsgrad für eine Geschäftsprozessanalyse und im weiteren Verlauf auch für die Erstellung eines unterstützenden Softwaresystems nicht ausreichend sein kann, folgt als nächstes die Dokumentierung der Einzelheiten der gefundenen Anwendungsfälle. Dabei wird wieder auf hybride Weise vorgegangen.

Textuell wird eine *essenzielle* Beschreibung des GAF angefertigt. Dies bedeutet vor allem den Verzicht auf Aspekte, die nichts mit

dem unmittelbaren geschäftlichen Zweck des GAF zu tun haben, sowie die vollständige Abstraktion von konkreten technologischen Rahmenbedingungen oder Lösungen (aus Sicht der MDA befinden uns zur Zeit ja auch in den Regionen des CIMs, wo solche Aspekte eh nichts zu suchen hätten).

Abb. 7.10 Essenzielle Beschreibung GAF Angebot erstellen

	Beschreibung Geschäftsanwendungsfall
Name	Angebot erstellen
Art	Kern-Geschäftsanwendungsfall
Kurzbeschreibung	Ein Kunde lässt sich ein Angebot für eine Versicherung von *M&M* erstellen.
Aktive Geschäftspartner	Kunde
Passive Geschäftspartner	keine
Innenorientierte Mitarbeiter	keine
Außenorientierte Mitarbeiter	Berater
Auslöser, ggf. Motivation	Ein Kunde wendet sich an *M&M*, um ein Angebot für einen Neuabschluss oder eine Änderung eines Versicherungsvertrages zu erhalten.
Ergebnis	Für den Kunden wurde ein Angebot für eine Versicherung erstellt.
Eingehende Daten	Angebotsanforderung
Vorbedingungen	keine
Nachbedingungen	keine
Essentieller Ablauf	▪ Produkttyp wählen ▪ Tarifrelevante Daten erfassen ▪ Tarife ermitteln ▪ Tarife vergleichen ▪ Tarife ins Angebot aufnehmen ▪ Kundendaten erfassen ▪ Angebotsdokument erstellen

Zur weiteren Verlängerung der „Haltbarkeit“ des Modells werden nur Sachverhalte modelliert und betrachtet, die sich in absehbarer Zeit, wenn überhaupt, nur unwesentlich ändern werden. Abb. 7.10 zeigt, wie eine solche Beschreibung für den bereits bekannten GAF `Angebot erstellen` aussehen kann.

7.2.9 Ablaufmodelle der Geschäftsanwendungsfälle

Wie angekündigt wurden die Geschäftsprozesse nicht nur textuell, sondern auch grafisch erfasst. Die so entstandenen Diagramme erfassen die dynamischen Teile der Geschäftsanwendungsfälle mittels UML-Aktivitätsdiagrammen. Abb. 7.11 zeigt eine Modellierung dieser Art auf grober Detailstufe.

Es gelten die Bedeutungen der Elemente eines Aktivitätsdiagramms (vgl. Anhang B). Im vorliegenden Modell wird zuerst der Produkttyp gewählt (z. B. Unfallversicherung, Kfz-Versicherung, …) und danach die tarifrelevanten Daten ermittelt.

Zu den erfassten Daten gehören etwa:

- die Risikomerkmale des Interessenten,
- die gewünschte Zahlweise,
- die erwartete Deckungshöhe,
- der vereinbarte Selbstbehalt,
- usw.

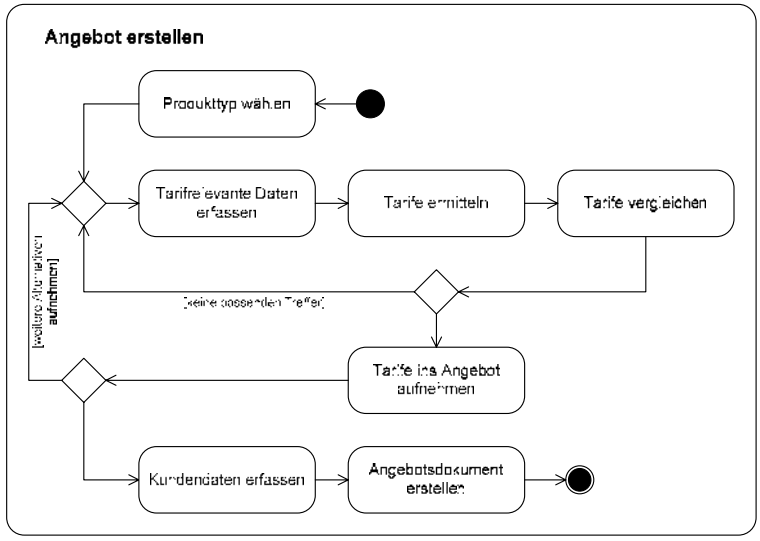


Abb. 7.11 Ablauf GAF Angebot erstellen

Auf Basis dieser Daten werden die möglichen Tarife ermittelt und vom Berater verglichen. Soweit sie den Anforderungen des Kunden genügen werden sie in das Angebot aufgenommen. Darüber hinaus

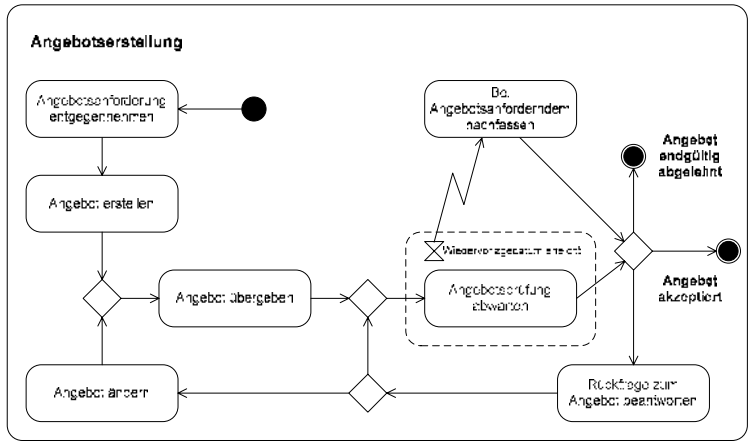
hat der Berater die Möglichkeit dem Angebot weitere Alternativen hinzuzufügen, indem er zusätzliche Tarife auf Basis veränderter Vorgabedaten in die Liste der zu erstellenden Angebote aufnimmt.

Ein ausreichender Detaillierungsgrad an dieser Stelle trägt wesentlich für die erfolgreiche Ableitung einer Ablaufmodellierung von eventuellen Systemanwendungsfällen aus den Ergebnissen der Geschäftsprozessanalyse bei. Also der Ableitung von Anwendungsfälle n einer den Geschäftsanwendungsfall unterstützenden Software. Die Erhöhung des Detaillierungsgrades des GAF wird auch im weiteren Verlauf des Fallbeispiels geschehen. An dieser Stelle wollen wir es jedoch beim gewählten Detaillevel belassen und uns stattdessen mit dem nächsten Schritt der GPA, der Ablaufmodellierung eines Geschäftsprozesses, befassen.

7.2.10 Ablauf Geschäftsprozess

Der in Abb. 7.12 dargestellte Geschäftsprozess `Angebotserstellung` zeigt den zuvor dargestellten Geschäftsanwendungsfall `Angebot erstellen` im Ablaufkontext mit anderen Geschäftsanwendungsfällen. Initiiert wird der Geschäftsprozess durch den GAF `Angebotsanforderung entgegennehmen`. Nach dem das Angebot erstellt und dem Kunden übergeben wurde muss zunächst die Angebotsprüfung des Kunden (hier Angebotsanfordernder genannt) abgewartet werden. Wird ein ggf. zuvor festgelegtes Wiedervorlagedatum überschritten, so ergreift unser Unternehmen die Initiative und fasst bei Angebotsanfordernden nach.

Abb. 7.12 Ablaufmodell Angebotserstellung



Dieser kann das Angebot akzeptieren, endgültig ablehnen oder aber Rückfragen stellen, die evtl. zu einer Änderung des Angebots füh-

ren. Das veränderte Angebot wird dann wieder dem Kunden übergeben, diese prüft es erneut und so weiter und so fort.

7.2.11 Geschäftsklassenmodell

Zur Erstellung des Geschäftsklassenmodells studieren wir unsere Kern-Geschäftsanwendungsfälle und suchen uns die darin verwendeten Geschäftsobjekte heraus.

Die Ermittlung der Geschäftsklassen erfolgt dabei durch die klassischen Mittel der Objektorientierten Analyse (OOA). Hier ist zum einen die „manuelle“, eigenständige Ermittlung der Geschäftsklassen durch Erfahrung und scharfes Hingucken, der Verwndung von Kategorielisten oder die Anwendung der „Substantivsuche“ in den entstandenen GPA-Ergebnissen möglich, wie sie beispielsweise in Craig Larmans *Applying UML and Patterns* vorgestellt wird [Lar04].

Ein weiterer Ansatz ist, sich die Erfahrung gestandener Analytiker zu Nutze zu machen und auf bestehende Sammlungen von Analyse-Mustern zurückzugreifen, wie sie beispielsweise in *Analysis Patterns : Reusable Object Models* von Martin Fowler [Fow96] oder in *Data Model Patterns: Conventions of Thought* von David Hay [Hay96] festgehalten sind.

Die mit der gewählten Methode gefundenen Geschäftsklassen werden dann in einem Klassendiagramm modelliert.

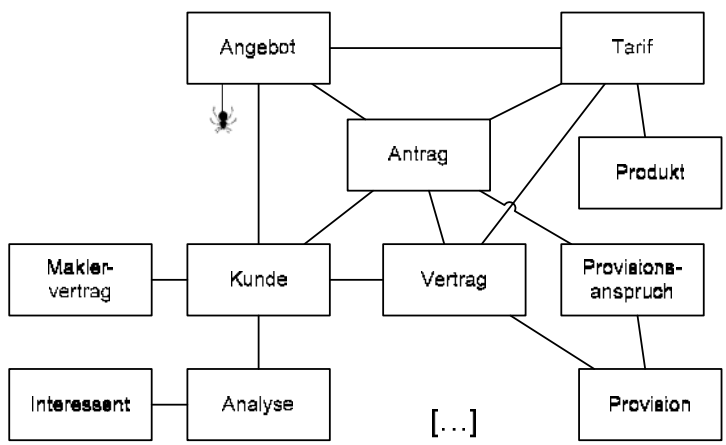


Abb. 7.13 Geschäftsklassen

Abb. 7.13 zeigt einen Ausschnitt aus einem solchen Diagramm. Der dargestellte Ausschnitt zeigt einen Teil des betrachteten, operativen

Bereichs unseres Modellierungsfokus, genauer von der `Analyse` über `Angebot` hin zu `Antrag` und abschließendem Versicherungsabschluss in Form eines `Vertrag`es.

Geschäftsklassenmodelle geben die strukturelle Beziehung der Geschäftsobjekte (bzw. -klassen) aus geschäftlicher Perspektive wieder. Auch dieses Modell behält seine Bedeutung im gesamten Projektzyklus bei. Es wird uns an späterer Stelle als Grundlage für Systemanalyse und -design in der anschließenden Systemerstellungsphase dienen.

7.3 Ergebnis der Geschäftsprozessmodellierung

Nachdem die Geschäftsprozesse des Modellierungsfokus im Ist-Zustand modelliert und – etwa in Form eines Prozesshandbuchs – dokumentiert worden sind, kann darauf aufbauend ein Maßnahmenkatalog abgeleitet werden.

In unserem Beispiel wurden für die Fixierung dieses Katalogs zwei Ausprägungen vorgesehen: zum einen eine detaillierte Version mit konkreten Handlungsanweisungen und zum anderen als Management-Summary so wie in Abb. 7.14 auszugsweise dargestellt.

Abb. 7.14 Management-Summary der GPA

[…]

Die bisherige Geschäftsanbahnung mit persönlichem Kontakt durch Berater und Makler sollte beibehalten und, parallel dazu, das Internet künftig als zusätzliche Vertriebsplattform genutzt werden.

Internet-Auftritt

Hierzu gehört zunächst die grundsätzliche Darstellung des Unternehmens im Internet selbst (etwa Team, Dienstleistungsportfolio, Kontakt) und die Definition fachbezogener werbewirksamer Inhalte. Dabei kann es sich beispielsweise um aktuelle Informationen zum Bereich Versicherung und Vorsorge handeln, also etwa zur Altersvorsorge (Stichwort Riester- bzw. Rürup-Rente). Weitere Ansatzpunkte wären das Anpreisen exklusiver Angebote oder der Hinweis auf besondere Termine (z. B. Kfz-Versicherungswechsel Stichtag).

Neben diesem eher statischen Anteil sollten dazu passende Services angeboten werden. In diesem Zuge haben wir die folgenden Geschäftsprozesse bzw. -anwendungsfälle als geeignet identifiziert:

Online-Angebot

(1) Der Geschäftsprozess «Angebotserstellung» sollte als Online-Version umgesetzt werden. Interessenten bzw. Kunden sollen dadurch in die Lage versetzt werden sich auf Basis ihrer persönlichen Bedarfe bzw. Vorgaben eigenständig ein Angebot von einem Online-System erstellen zu lassen. Sollte ein so erstelltes Angebot Akzeptanz beim Interessenten finden, so kann er ebenfalls online einen entsprechenden Antrag dazu stellen.

Online-Antrag, Workflow Management, B2B

(2) Die elektronische Weiterverarbeitung eines solchen Online-Antrages innerhalb des Geschäftsprozesses «Antrags-/Vertragsbearbeitung» stellt unserer Ansicht nach die logische Fortführung dazu dar. Die Umsetzung sehen wir in zwei Teilpaketen:

- zum einen die Vorgangssteuerung der Anteile des Geschäftsprozesses die bei *M&M* beheimatet sind über ein Worfklow-Management System
- und zum anderen die volle Automatisierung der Abläufe zwischen *M&M* und den Produktgebern (B2B-Integration).

Das zuletzt genannte Paket (2) sollte dabei, aufbauend auf der ersten Phase, in einer zweiten Stufe realisiert werden.

Portaldienste, Customer Self Services

In einer weiteren Stufe empfehlen wir den Ausbau der Kundenselbstbedienungsdienste (Customer Self Services, CSS) zu einem Kundenportal. Für den Bereich «Partnerverwaltung» ist hier zuvorderst der Geschäftsanwendungsfall «Kundendaten verwalten» (Änderung einer Adresse oder einer Bankverbindung) zu nennen. Dies geschieht zuvorderst, um das Call-Center zu entlasten, sollte aber auch in Hinsicht auf mögliche Folgeaktivitäten betrachtet werden. Beispielsweise könnte ein solches Portal als Ausgangspunkt zum Sammeln von Akquisedaten gesehen werden.

[…]

7.4 Das weitere Vorgehen

Da das Projekt mit MDA-Mitteln durchgeführt werden soll, müssen nicht nur die Anforderungen an das System ermittelt und umgesetzt werden, die aus der eben beschriebenen Geschäftsprozessanalyse und den gegebenen Empfehlungen resultieren, sondern ebenfalls die

Anforderungen an die MDA-Infrastruktur ermittelt und die benötigten Werkzeuge implementiert werden.

Verzichtet werden soll dabei auf die Verwendung kommerziell-, proprietärer Software zugunsten offener Standard und frei verfügbarer Open-Source (OS) Technologien. Dabei sind auch Überlegungen zu den unterschiedlichen Lizenzen von OS anzustellen.

Insgesamt also eine Menge Aufgaben, die zu erledigen sind. Im nächsten Kapitel werden die ersten Phasen des Projektes zusammengefasst, in denen es um die Planung, Vorbereitung, Anforderungserfassung sowie die Erstellung erster Technologiepläne geht. Im weiteren Verlauf des Buchs werden wir dann die Realisierungsphase in Augenschein nehmen.

Literatur

[CK92] Michael G. Christel und Kyo C. Kang: *Issues in Requirements Elicitation*. Carnegie Mellon University/Software Engineering Institute, 1992.
http://www.sei.cmu.edu/publications/documents/92.reports/92.tr.012.html (letzter Abruf Januar 2006)

[Fow96] Martin Fowler: *Analysis Patterns – Reusable Object Models*. Addison-Wesley, 1996.

[Hay96] David C. Hay: *Data Model Patterns – Conventions of Thought*. Dorset House, 1996.

[Lar04] Craig Larman: *Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd Edition*. Prentice Hall, 2004.

[OWS+03] Bernd Oesterreich, Christian Weiss, Claudia Schröder, Tim Weilkiens und Alexander Lenhard: *Objektorientierte Geschäftsprozessmodellierung mit der UML*. dpunkt, 2003.

8 Projektplanung

„Ja, mach nur einen Plan, Sei nur ein großes Licht! Und mach dann noch 'nen zweiten Plan. Gehn tun sie beide nicht."

Berthold Brecht (Dreigroschenoper)

Das vorliegende Kapitel führt unser fiktives Projekt mit der Beschreibung der Projektplanung weiter.

8.1 Exploratory 360º

Zu Beginn des Projektes wird ein initialer Rundumblick dokumentiert, der die wichtigsten Eckdaten des Projektes absteckt. Die Erstellung dieses, *Exploratory 360°* genannten, Artefaktes steckt die wesentlichen Eckdaten des Projektes fest. Zu diesen zählen zum Beispiel auch die Folgenden:

- der *Business-Case*, der klarstellt, warum sich das geplante Vorhaben überhaupt lohnt,
- die *Vision* des zu erstellenden Systems sowie die ersten *High-level Anforderungen*, die an dieses System gestellt werden,
- *Technologie-Pläne*, die erste Lösungsansätze für bereits bekannte Probleme liefern, und damit die grundsätzliche Machbarkeit des Vorhabens beweisen,
- die *Team-Zusammensetzung (engl. staffing)* und letztendlich
- ein Überblick über das benutzte *Prozessmodell*

Auf die Anführung eines Business-Case sowie die detaillierte Erläuterung des benutzten Prozessmodells und des Projektstaffings soll allerdings an dieser Stelle verzichtet werden. Tipps und detail-

lierte Betrachtungen zu diesen Themen kamen in Kapitel 6 ausführlich zur Sprache und sollen hier nicht wiederholt werden.

8.1.1 Systemanforderungen

Überspringen wir also diesen Schritt und machen sofort mit der Erhebung der Vision sowie erster Anforderungen, die an das zu erstellende System gestellt werden, weiter.

Systemidee/Vision

Die Prozesse der *M&M* sollen durch ein web-basiertes Kundenportal unterstützt und werden, das folgende Funktionalität realisieren soll:

- das *Ändern von Stammdaten* durch den Kunden selbst,
- die *Bereitstellung von Informationen* zu Produkten und rechtlichen Aspekten der Vorsorge,
- die *Anbahnung von Kontakten* zwischen Interessenten und einem zuständigem Vermittler,
- die *automatische Erstellung eines Angebotes* angestoßen durch den Interessenten. Dabei soll aufgrund des pilotierenden Charakters des Vorhabens vorerst nur die relativ einfache Produktkategorie der *Tierhalterhaftpflicht* angeboten werden. Die Software nutzt den bereits vorhandenen Rechenkern des Unternehmens, um nach der Ermittlung der notwendigen Daten konkrete Produktkombinationen vorzuschlagen, den identifizierten Bedarf abzudecken und letztlich dem Interessenten die Tarife der Vorschläge anzuzeigen.
- Letztlich soll es möglich sein auf Basis eines erfolgten Angebotes entweder den Anstoß zur Erstellung eines *detaillierteren schriftlichen Angebots* zu erteilen oder *direkt auf elektronischem Wege einen Antrag* auf die Erteilung des gewählten Versicherungsschutzes zu stellen.

Systemanwendungsfälle

Aus der im vorigen Abschnitt erläuterten Systemidee und der Geschäftsprozessanalyse (GPA) im vorigen Kapitel ergeben sich die ersten Anwendungsfälle des Systems. Bei der Durchführung der erwähnten GPA wurde der Geschäftsanwendungsfall `Angebot erstellen` als Kernanwendungsfall identifiziert. Die Durchführung dieses Vergleiches online über das Internet (hier `Online-Tarifcheck` genannt) ist eine mögliche Realisierung

dieses GAF durch einen Systemanwendungsfall (SAF). Der Sachverhalt wird durch das folgende Diagramm ausgedrückt (
Abb. 8.1):

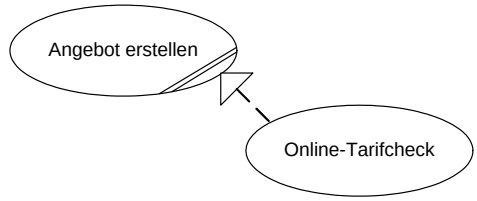


Abb. 8.1 Use Case: SAF Online-Tarifcheck

Konvergentes Vorgehen

Das Vorgehen verdeutlicht die Vorteile einer Vorschaltung der Geschäftsprozess- vor die eigentliche Systemanalyse: Ergebnisse werden direkt wiederverwendet, sodass bereits geleistete Arbeit nicht verloren geht. Systemanwendungsfälle realisieren Geschäftsanwendungsfälle, Geschäftsklassen dienen als Grundlage der Systemanalyse und des Designs der technischen Daten- und Klassenmodellierung.

	Beschreibung Systemanwendungsfall
Name	Online-Tarifcheck
Kurzbeschreibung	Die Produkte einer bestimmten Versicherungsart werden auf Basis der individuellen Risikomerkmale des Interessenten automatisiert miteinander verglichen.
Akteure	Interessent
Auslöser, ggf. Motivation	Ein Interessent surft auf die Seite des Portals, um einen Online-Vergleich durchzuführen.
Vorbedingungen	keine
Eingehende Informationen	keine
Ergebnis(se)	Die in Frage kommenden Produkte und die entsprechenden Tarife wurden ermittelt. Die Ergebnisse werden dem Interessenten angezeigt. Der Interessent kann einzelne Angebote auswählen und ein individuelles Angebot beauftragen, bzw. direkt online den Antrag auf eine Versicherungspolice stellen.

Abb. 8.2 SAF Online-Tarifcheck

Auch Abläufe von Geschäftsprozessen, die in der Geschäftsprozessanalyse modelliert wurden, können bei der Erstellung der unterstüt-

zenden Software genutzt werden. Dies gilt ebenso für Akteur- und Rollenmodelle, da die Benutzer des Systems im Wesentlichen aus den Reihen der Personen stammen, die zuvor in der GPA bereits identifiziert worden sind.

Nachsicht …

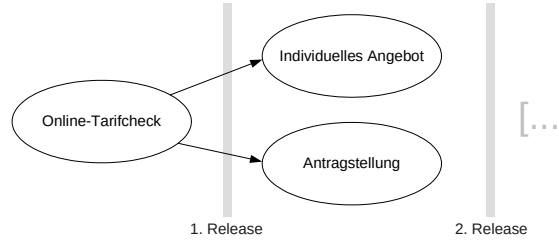
Da wir in diesem Teil des Buches die technischen Aspekte des MDA-Ansatzes in den Vordergrund stellen wollen, verzichten wir an dieser Stelle auf die detailliertere Beschreibung der Systemanforderungen und hoffen, dass nach den Beschreibungen im vorigen Abschnitt sowie der Einführung des Fallbeispiels im vorigen Kapitel zumindest intuitiv erahnbar ist, was produziert werden soll. Im nächsten Kapitel werden wir trotzdem die Anforderungsanalyse als abgeschlossen voraussetzen, um mit der Beschreibung des Projektverlaufes fortfahren zu können.

8.1.2 Ein erster Releaseplan der Anwendung

Die fertige Anwendung zur Durchführung des web-basierten Online-Tarifchecks soll bereits mit der ersten Auslieferung an den Kunden übergeben werden. Das Projekt wird zudem als *Pilotprojekt* (vgl. Ausführungen zu diesem Thema in Kapitel 6) zur Erstellung des genua MDA-Frameworks konzipiert. Das Projekt teilt sich so in zwei Hauptstränge: die Entwicklung des Frameworks an sich sowie die Entwicklung des Softwaresystems unter Verwendung der so entstandenen Elemente des Frameworks.

Der vorläufige Releaseplan des Systems ist in Abb. 8.3 dargestellt. Die Pfeile zwischen einzelnen Releases sind im Sinne einer Abhängigkeit in umgekehrter Pfeilrichtung zu verstehen:

Abb. 8.3 Releaseplan des Systems



Die Funktionalität `Individuelles Angebot` kann erst fertig gestellt werden, wenn der `Online-Tarifcheck` fertig gestellt ist, da individuelle Angebote nur auf Basis zuvor ermittelter, und vom Kunden akzeptierter, Angebote durchgeführt werden können. Gleiches gilt für die `Antragstellung`, da auch online gestellte Anträge auf die Auswahl eines Ergebnisses des Tarifchecks folgen.

Darstellung und Konzept des dargestellten Plans, wie auch die Technik des Exploratory 360°, sind der *Crystal Clear* Methode von Alistair Cockburn entnommen. Wer an näheren Einzelheiten interessiert ist, findet diese unter [Coc04].

Crystal Clear

Wie bereits angedeutet soll die „Beobachtung“ des Projektes mit Abschluss des ersten Releases enden, die Anforderungen zur Auslieferung 2, die zusätzlich benötigte Module zur Angebots- und Antragstellung enthält, werden daher nicht näher betrachtet.

8.1.3 Anforderungen an „genua“

Nachdem im Groben klar ist, welches System wir für den Kunden bauen wollen, ist es nun Zeit sich mit dem System für die Entwickler zu beschäftigen, das im gewählten Ansatz fast ebenso wichtig ist. Im folgenden Abschnitt sind die Anforderungen an die geplante MDA-Infrastruktur aufgeführt, mit deren Werkzeugen wir in Zukunft Applikationen bauen wollen. Wir wollen die Sammlung aller Assets, die zur Infrastruktur gehören im Folgenden *genua* nennen, einer Abkürzung, die ursprünglich etwas mit der „**Gen**erierung von **u** **A**nwendungen“ zu tun hatte. Was das „u“ bedeutet, haben wir leider zwischenzeitlich vergessen. Wer eine gute Idee hat, kann uns natürlich schreiben, wir werden dann die beste Einsendung in den folgenden Auflagen berücksichtigen ;o).

Systemidee

genua soll also die Entwicklung eines Softwaresystems unter Nutzung eines MDA-basierten Ansatzes ermöglichen. Die Hauptfunktionalitäten des Frameworks gliedern sich dabei in die folgenden Bereiche (Abb. 8.4).

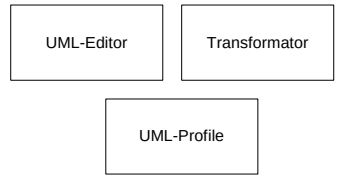


Abb. 8.4 Funktionsbereiche des genua Frameworks

Tools of the trade

Ein rudimentärer *UML-Editor*: Die Erstellung eines solchen Editors wird deshalb notwendig, da zum aktuellen Zeitpunkt kein kommerzieller Editor mit der benötigten Funktionalität und akzeptablen ROI zur Verfügung steht. Die übrige Architektur ist dennoch so zu gestalten, dass das Framework grundsätzlich nicht abhängig vom verwendeten Editors ist. Dies gilt vor allem für den *Transformator*.

Dieser muss in der Lage sein *Model-zu-Model (M2M)* sowie *Model-zu-Text (M2T)* Transformationen durchzuführen. Diese Transformationen müssen zudem in geeigneter Form beschrieben werden können. Die *Beschreibungssprache*, die hierzu benötigt wird, gehört ebenfalls zum geforderten Funktionsumfang des Frameworks.

Letzter Punkt dieser vorläufigen Aufzählung der High-level Anforderungen ist die Erstellung geeigneter *UML-Profile* zur Modellierung der zu erstellenden Systeme. Initial sollen die folgenden Domänen abgedeckt werden:

- Persistenz
- Dialogfluss
- Masken sowie
- Workflow

Die Profile sollen zur Unterstützung des abstrahierenden Gedankens von MDA, sowohl plattform-abhängige als auch -unabhängige Modellierung unterstützen. Es handelt sich um insgesamt also acht Profile, die im weiteren Verlauf des Pilotprojektes entwickelt werden müssen.

Kernanforderungen an das Framework

Basierend auf den Überlegungen der Systemidee, ergeben sich folgende Kernaufgaben, die im Projektbereich der Framework-Erstellung zu erledigen sind.

Beginnen wir mit den Aufgaben rund um das Thema der *Editor* Erstellung (Abb. 8.5) . Der mit `ID 1` versehene Task zur Evaluierung der zur Umsetzung benötigten Technologien beginnt schon in der Projektplanungsphase (ab Abschnitt 8.2), weshalb wir zur näheren Erläuterung an diese Stelle verweisen und sofort mit den folgenden Punkten fortfahren wollen. Funktional soll der Editor in der Lage sein, Modelle, Profile und Metamodelle für Klassen- sowie Aktivitätsdiagramme zu erstellen und diese sowohl laden als auch speichern zu können. Dabei sollen – hier nicht näher aufgeführt – die notwendigen, handelsüblichen Funktionen zum Editieren der Modelle unterstützt werden. Wichtig an dieser Stelle ist festzuhalten, dass initial lediglich ein *rudimentärer* Editor angestrebt ist, der so früh wie möglich durch eine geeignete kommerzielle Variante ersetzt werden soll.

ID	Beschreibung
1	Technologie-Evaluierung
2	Modelle / Profile / Metamodelle
2.1	Klassendiagramme
2.2	Aktivitätsdiagramme
3	Modelle laden
3.1	Eigenformat
3.2	XML Metadata Interchange (XMI)
4	Modelle speichern
4.1	Eigenformat
4.2	XML Metadata Interchange (XMI)
4.3	Diagram Interchange (DI)

Abb. 8.5 Kernaufgaben Editor

Widmen wir uns nun der zweiten Hauptkomponente genuas. Zur Erstellung des *Transformators* stehen folgenden Aufgaben an (Abb. 8.6).

ID	Beschreibung
5	Transformationskonzept
5.1	Beschreibungssprache
5.2	Model-to-Model Transformation (M2M)
5.3	Model-to-Text Transformation (M2T)
6	Technologie-Evaluierung
7	Implementierung des Transformators

Abb. 8.6 Kernaufgaben Transformation

Hervorzuheben ist hier die Erstellung eines *Transformationskonzeptes* (`ID 5`). Die hier anstehenden Probleme sind im Kapitel über

Transformationen (Kapitel 5) erläutert. Nachdem die zur Umsetzung notwendigen Technologien gefunden und evaluiert sind, kann dann die tatsächliche Realisierung/Implementierung des Transformators in Angriff genommen werden. Hierbei sind dann auch die Detailanforderungen an den Transformator festzulegen.

Abb. 8.7 Kernaufgaben Profile

ID	Beschreibung
8	Persistenzkonzept
9	Persistenzprofil
9.1	plattform-unabhängig (PI)
9.2	plattform-spezifisch (PS)
10	Konzept Dialogfluss
11	Profil für Dialogfluss
11.1	plattform-unabhängig (PI)
11.2	plattform-spezifisch (PS)
12	Konzept Masken
13	Profil Masken
13.1	plattform-unabhängig (PI)
13.2	plattform-spezifisch (PS)
14	Konzept Workflow
15	Profil Workflow
15.1	plattform-unabhängig (PI)
15.2	plattform-spezifisch (PS)

Der Überblick über die initialen Aufgaben wird durch die Angabe der zu erstellenden *Profile* (Abb. 8.7) abgeschlossen. Wir wollen hier *Bottom-Up* vorgehen und zuerst ein Konzept zur technischen Realisierung der zu abstrahierenden Aspekte (*Persistenz*, *Dialogfluss*, *Masken*, *Workflow*) erstellen, und erst dann an die Erstellung der tatsächlichen Profile gehen. Dies bedeutet also, dass wir uns

zuerst eine (erste) Plattform aussuchen, und mit diesem Wissen die Profile erstellen.

8.1.4 Releaseplan des genua Frameworks

Die Realisierung oben genannter Funktionen sieht in einem ersten Plan wie folgt aus (Abb. 8.8):

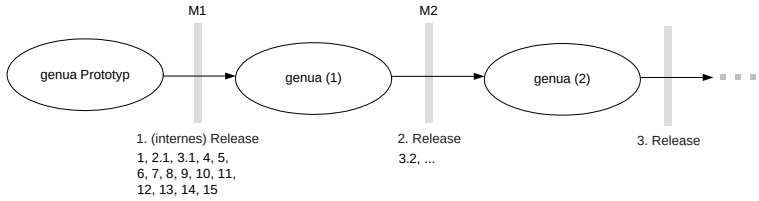


Abb. 8.8 Releaseplan genua Framework

Die Realisierung des genua Prototyps umfasst im ersten Release (Milestone `M1`) nur die zur ersten Arbeit mit der Infrastruktur unbedingt notwendigen Aufgaben Die im Pilotprojekt gesammelten Erfahrungen bis `M2` sollen dann sukzessive ins Framework eingehen sowie auch eventuell zusätzliche, an dieser Stellen nicht aufgeführte, Anforderungen geprüft und realisiert werden. Die ständige Weiterentwicklung und Verbesserung des Frameworks ist ein ständiges Ziel, das im Folgenden, obwohl nicht immer explizit aufgeführt, im Hinterkopf behalten werden muss.

8.1.5 Erste Projektpläne

Kombiniert man die Releasepläne der beiden Teilprojekte ergibt sich folgendes Bild (Abb. 8.9):

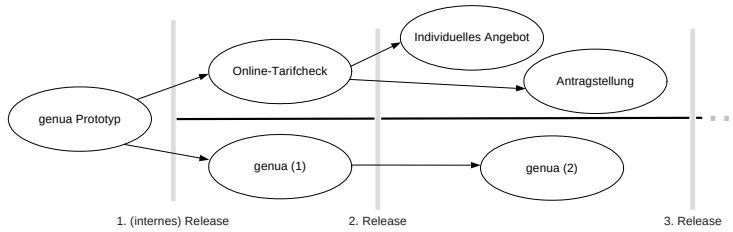


Abb. 8.9 Releaseplan des Gesamtprojektes

Ein Kernteam erstellt einen genua Prototyp, der dem ersten geplanten Release entspricht. Mit den dort erstellten Werkzeugen und Verfahrensweisen wird dann der `Online-Tarifcheck` realisiert und das Framework weiter optimiert und verbessert. Jedes weitere

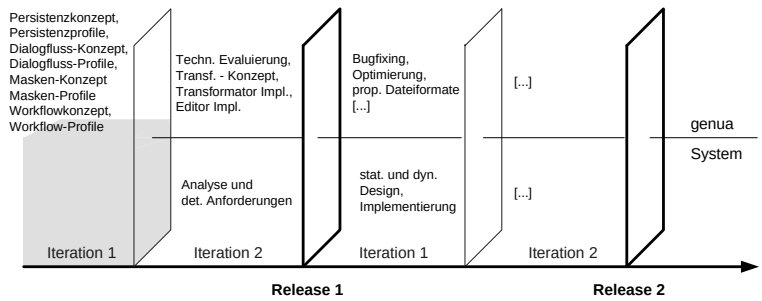
Release baut dann auf diesen Ergebnissen auf und verfeinert sie weiter.

Die Funktionalitäten, die bei Erreichen des jeweiligen Milestones abgedeckt werden müssen, ergeben sich aus den Einzel-Releaseplänen der beiden Teilprojekte. Zu beachten ist, dass sich die Nummerierung der Releases beim Softwaresystem um eins nach „rechts“ verschiebt, da das Initialrelease des genua Frameworks nicht mit der Entwicklung des der `Online-Tarifchecks` parallel gestartet werden kann bzw. auf jeden Fall eine Vorlaufzeit benötigt.

Deutlich wird dies auch im vorläufigen *Iterationsplan* (Abb. 8.10). Auf der Projektverlaufsabszisse dargestellt sind in der oberen Hälfte die Iterationsziele für die Entwicklung des genua Frameworks, während in der unteren Hälfte entsprechend die Ziele für das Anwendungssystem angeführt werden.

Angestrebt ist die Fertigstellung der Konzepte für die technischen Querschnittskomponenten sowie die Erstellung der zugehörigen Profile in der ersten Iteration durch das Kernteam, bevor die Entwicklung des Systems durch die Mitglieder des Pilotprojekts beginnt. Dies geschieht vor allem vor dem Hintergrund, dass die Analyse und die Erhebung der Anforderungen des Systems in Iteration 2 schon mit den gerade genannten Artefakten modelliert werden können.

Abb. 8.10 Vorläufiger Gesamt-iterationsplan



Die Entwicklung des genua Prototyps endet mit Release 1. Zu diesem Zeitpunkt stehen dann genug Funktionen zur Verfügung, um mit der Entwicklung des Systems zu beginnen. Das Kernteam der genua Realisierung bleibt während der folgenden Iterationen vollständig erhalten und verbessert das Framework nach den Angaben und Erfahrungen des Anwendungsentwicklungs-Teams. Gleichzeitig können so die mit Bestimmtheit auftretenden Fehler im Framework zeitnah durch Hot-Fixes beseitigt, und so eventuell auftretende Probleme in den Griff bekommen werden.

8.2 Technologie-Plan

An dieser Stelle werden einige Schlüsseltechnologien vorgestellt, die wesentlich zur Realisierung der gesteckten Anforderungen des Frameworks sind. Sie stellen einen Teil der geforderten „Konzepte“ dar, indem sie die Technologien zur Umsetzung der geforderten Features des genua-Frameworks festlegen. Der zweite Teil, die tatsächliche Umsetzung der Anforderungen mittels dieser Technologien, findet in der Projektdurchführung statt, die uns im nächsten Kapitel beschäftigen wird.

Um Redundanzen zu vermeiden, sind an dieser Stelle jedoch nicht alle benutzten Technologien aufgeführt. Sie tauchen an der Stelle im Projektverlauf auf, an der ihre Verwendung motiviert ist, und werden auch dort im Detail beschrieben.

8.2.1 Das Eclipse-Projekt

Da viele der verwendeten Technologien unter dem Mantel des Eclipse Projektes beheimatet sind wird an dieser Stelle eine kurze Einführung in die Geschichte und den grundsätzlichen Aufbau gegeben. An späterer Stelle sind dann die konkret benutzten Teile des Projektes im Framework aufgeführt.

Ursprünglich als internes Projekt konzipiert, wurde Eclipse von IBM nach einigen Investitionen (~40 Millionen USD) unter die CPL (Common Public License) gestellt und zur weiteren Entwicklung an ein Open Source Konsortium übergeben, unter dessen Führung im November 2001 die Version 1.0 erschien. Gründungsmitglieder dieses Konsortiums waren: Borland, Merant, QNX Software Systems, Rational Software, Red Hat, SuSE, TogetherSoft und Webgain. Die konzeptionelle Führung behielt jedoch weiterhin IBM, vertreten durch ihre Tochterfirma Object Technologies International (OTI), den ursprünglichen Entwicklern der Plattform.

Die Gesamtführung untersteht dem so genannten „Board of Stewards“, das aus einem Repräsentanten pro Gesellschafter besteht, und das die Erreichung des Gesamtziels - eine gesunde Balance zwischen Open Source- und Kommerziellen Interessen seiner Mitglieder – überwacht, sowie auf technischer Seite dem Project Management Comitee (PMC) das seinerseits die Koordination der Subprojekte (wie beispielsweise Java Entwicklungsumgebung, Plattform, …) und deren Komponenten auf operationeller Ebene überwacht.

Die Common Public License (CPL)

Als Projekt, das den Anspruch hat, kommerzielle Interessen sowie den Open Source Gedanken gleichermaßen zu berücksichtigen, werden alle Bestandteile der Eclipse Plattform unter die *Common Public License (CPL)* [CPL10] gestellt. Dies bedeutet vor allem, dass Produkte, die unter Nutzung von Eclipse-Technologien entstanden sind, unter eigener Lizenz kommerziell genutzt und unter eigenen Lizenzbestimmungen binär veröffentlicht werden dürfen. Sie garantiert den Unternehmen den Schutz des geistigen Eigentums an den entstandenen Produkten, und steht den so genannten viralen Lizenzen gegenüber, die die Veröffentlichung abgeleiteter Produkte unter der gleichen Lizenz erzwingen, denen die verwendeten Komponenten unterstehen. Bei Nutzung von Open Source Komponenten würde dies die Veröffentlichung der entstandenen Produkte ebenfalls unter Open Source Lizenz bedeuten, was sie damit in vielen Fällen für kommerzielle Anwendung uninteressant macht. Da diese Art Lizenzen dem traditionellen Copyright gegenüberstehen heißen sie auch *Copyleft Lizenzen*. Wie erwähnt, besitzt die CPL diesen viralen Charakter nicht, und eignet sich daher uneingeschränkt zum kommerziellen Einsatz.

Näheres zu Open Source und der CPL findet sich auf [OSI] und [CPLFAQ], Betrachtungen zum Copyleft Charakter bestimmter Lizenzen werden in [Copyleft] angestellt.

Eclipse-Architektur

Charakteristisch ist der modulare Aufbau von Eclipse, der in Abb. 8.11 dargestellt ist.

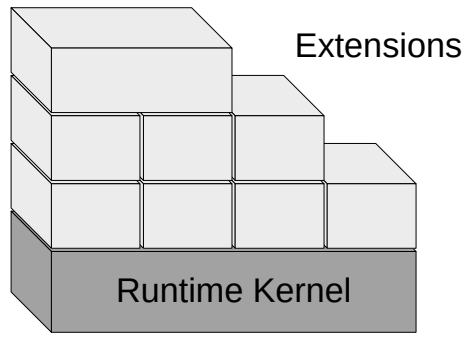


Abb. 8.11 Modularer Aufbau des Eclipse Projektes

Während viele Entwicklungsumgebungen die Möglichkeit bieten vorhandene Funktionalität durch eigene Module zu erweitern, ist Eclipse nach dem Prinzip entwickelt, dass lediglich eine Basis

unterstützender Dienste und Komponenten zum festen Kern von Eclipse gehören (die so genannte *Eclipse Plattform*), während alle konkreten Anwendungen (Java Entwicklungsumgebung, Debugger, CVS Client, …) als *Erweiterungen (engl. extensions)* der Plattform angesehen werden, die an definierten Erweiterungspunkten – den so genannten *Extension Points* – wiederum selbst als Grundlage neuer Komponenten dienen können. Eclipse bietet damit nicht nur die Möglichkeit einer horizontalen Erweiterung, sondern ermöglicht durch das unterliegende Konzept auch den vertikalen (auf bestehende Komponenten aufbauenden) Ausbau der Umgebung.

Abb. 8.12 verdeutlicht dies an der Darstellung der Standard Eclipse Distribution in drei Schichten. Zuunterst die *Eclipse-Plattform*, die die Basis aller Eclipse basierten Anwendungen darstellt, und seit Version 3 auch unter der Bezeichnung *Rich-Client-Plattform (RCP)* bekannt ist. Hervorgehoben auf der linken Seite die Runtime-Komponente, die die Erweiterungen (in Form so genannter Plug-ins) lädt und diese in die Arbeitsumgebung integriert. Das Nachladen geschieht dabei *Just-in-time*, wenn das Plug-in tatsächlich benötigt wird. Beim Start werden nur Metadaten der Erweiterungen eingelesen, die in einem Deskriptor abgelegt sind, was das Skalieren der Umgebung auch bei einer großen Anzahl von Erweiterungen erlaubt.

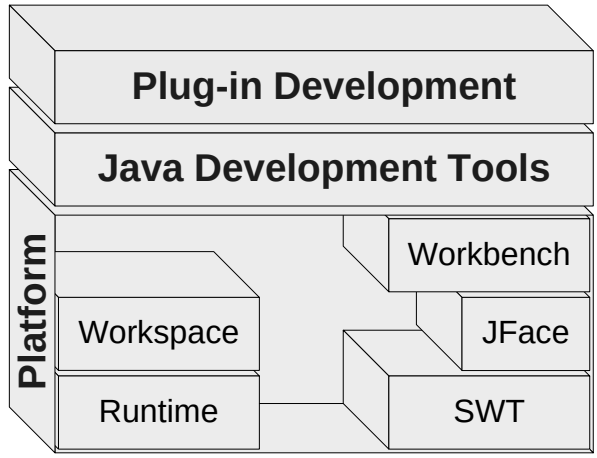


Abb. 8.12 Drei Schichten Modell der Eclipse Standard Distribution

Auf der rechten Seite der Abbildung finden sich die Bausteine, die Eclipse zur Entwicklung grafischer Benutzerschnittstellen zur Verfügung stellt: Das Standard-Widget-Toolkit (SWT), eine Klassenbibliothek, die Zugriff auf das nativ zugrunde liegende Fenstersystem bietet, wodurch sich Oberflächen, die mit SWT erstellt

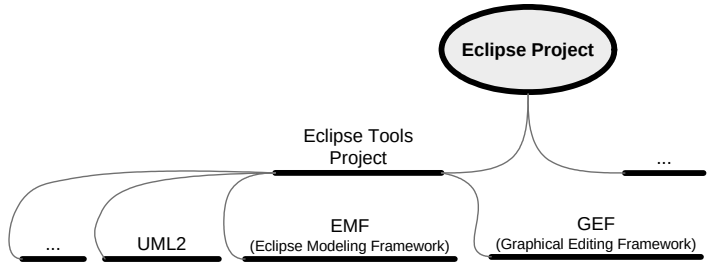
wurden nicht nur genau so aussehen und „anfühlen“ (*look-and-feel*) wie die Bedienelemente des unterliegenden Betriebssystems, sondern tatsächlich auch dessen native und oftmals hochoptimierte Implementierung dieser Widgets nutzt. Des Weiteren ist das auf SWT aufbauende JFace dargestellt, das Standardimplementierung für häufig benötigte GUI-Funktionalität bereitstellt (z. B. Dialoge, Wizards, etc.). Weitere Kernkomponente: die Workbench, die die Standard-Elemente einer Eclipse Oberfläche definiert, zu der Elemente wie Editoren, Sichten (engl. views) und Perspektiven gehören, die durch eigene Funktionalitäten erweitert, das spätere Aussehen der Arbeitsumgebung bestimmen.

Das Eclipse SDK schließlich liefert zu der Plattform die zwei Standard Plug-ins des Projektes mit: das Java Development Toolkit (JDT), das die Java IDE von Eclipse inklusive Source Debugger darstellt sowie das Plug-in Development Environment (PDE), das Wizards und unterstützende Werkzeuge zur Erstellung eigener Plug-ins zur Verfügung stellt.

Das Eclipse Tools Projekt (ETP)

Ein für die Realisierung des genua Frameworks besonders wichtiges Subprojekt von Eclipse ist das *Eclipse Tools Projekt* (Abb. 8.13).

Abb. 8.13 Übersicht Eclipse Tools Project



Eine ausführliche Übersicht über dieses Subprojekt findet sich unter [ETP]. Hier finden sich Werkzeuge und Bibliotheken, die den Entwicklern von Applikationen ihre Arbeit erleichtern sollen. Da Eclipse besonders die Unterstützung der Entwicklung (kommerzieller) Entwicklungswerkzeuge im Auge hat, befinden sich gerade in dem für uns besonders interessanten Bereich modellbasierte Entwicklung, Codegenerierung, Erstellung (visueller) Editoren für DSLs hier Werkzeuge, die für die Entwicklung von MDA-Werkzeugen wie geschaffen scheinen. Tatsächlich ist die Integration der Projekte unter dem Aspekt der Schaffung einer MDA-Infrastruktur erklärtes Ziel der ETPs.

Eclipse im genua Framework

Neben der Eclipse Entwicklungsumgebung spielen folgende Eclipse Tools Schlüsselrollen im Framework:

- ***EMF*:** ist vereinfacht ausgedrückt ein Modellierungs-Framework mit Fähigkeiten zur Codegenerierung unter Eclipse. Dass diese Beschreibung des Eclipse Modeling Frameworks eine starke Untertreibung darstellt, wird in der ausführlicheren Beschreibung im weiteren Verlauf dieses Kapitels deutlich.
- ***UML2*:** baut auf EMF auf, und stellt die Java-Implementierung des UML 2.x Metamodells dar. Auch die Verwendung dieses Frameworks wird an späterer Stelle ausführlich beschrieben.
- ***GEF*:** unterstützt die grafische Darstellung eines Modells, sowie die Möglichkeit dieses auch grafisch zu editieren. Auch hierzu später mehr.

Literaturempfehlungen zum Einstieg

Ersten Überblick über das Eclipse Projekt und die Eclipse IDE bietet *Eclipse in Action* [GBM03]. Alles Wissenswerte zur Erstellung eigener Erweiterungen kommt von Gamma und Beck persönlich mit *Contributing to Eclipse - Principles, Patterns und Plug-Ins* [GB03]. Das *Eclipse-FAQ 3.0* kann unter *http://www.eclipsefaq.org* konsultiert werden. Diese Sammlung ist auch in gedruckter Form zu haben [AL04].

Literaturempfehlungen zu benutzten Unterprojekten der Eclipse Familie werden an separater Stelle gegeben, generell können jedoch alle Bücher der offiziellen *Eclipse Series* empfohlen werden.

8.2.2 Hibernate

Als Plattform zur technischen Realisierung des Persistenz-Aspektes haben wir uns für das *Hibernate Framework* entschieden. Die Persistierung geschieht in Hibernate durch einen Mechanismus, der Objekt-Relationales Mapping (engl. Object/Relational Mapping, ORM) genannt wird. Dabei werden die Attribute von Java-Klassen für den Benutzer transparent auf Datenbank-Strukturen abgebildet. Diese Mapping genannte Abbildung wird durch die Angabe von Metadaten ermöglicht, die in Form von Mapping-Datei(en) dem Framework zugänglich gemacht werden.

Hibernate kann sowohl stand-alone, als auch in Umgebungen (Containern/Environment) ausgeführt werden. Bei der container-basierten

Ausführung sind die von Hibernate übernommenen Dienste davon abhängig, ob es sich um ein *managed* oder *non-managed Environment* handelt. Dabei fallen Servlet-Container wie Tomcat unter die zweite Kategorie, während die erste vollständigen Applikationsservern wie Websphere, WebLogic o. Ä. vorbehalten bleibt. In dieser Publikation wollen wir uns auf die Verwendung von non-managed Umgebungen beschränken. Abb. 8.14 zeigt Hibernate in diesem Umfeld.

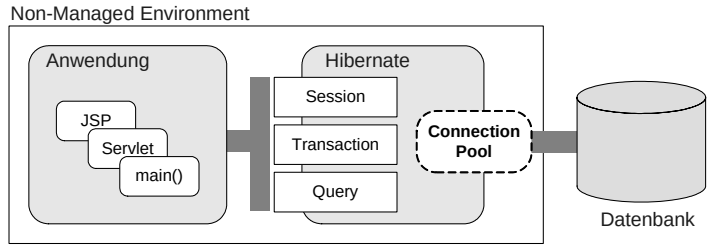


Abb. 8.14 Hibernate Architektur (nach [BK04])

Anwendungen greifen über die Schnittstellen `Session`, `Transaction` und `Query` auf die angebotenen Dienste zu, während Hibernate alle Querschnittsaufgaben, wie beispielsweise das Pooling von Connections, das Cachen von Daten und Queries, etc. übernimmt bzw. an weitere zuständige Services weiterleitet.

Da im Folgenden bei der Erstellung der Persistenzprofile sowie der eigentlichen Transformation selbst, Besonderheiten dieser Technologie eine Rolle spielen, soll anhand eines kurzen Beispiels ein Eindruck davon vermittelt werden, wie sich der Umgang mit Hibernate „anfühlt“ bzw. welche Arten von Artefakten und Überlegungen beim weiteren Vorgehen eine Rolle spielen. Am Ende dieses Abschnittes wird dann auf weiterführende Literatur verwiesen, die eine intensivere Beschäftigung mit der Materie ermöglicht.

Installation

Wer das folgende Beispiel selbst ausprobieren möchte, hat mit der Installation von Hibernate zum Glück keine zeitraubende Prozedur vor sich:

- **Schritt 1:** Herunterladen der aktuellen Hibernate Distribution von *www.hibernate.org*. Zur Zeit der Drucklegung ist dies die Version 3.0.5.
- **Schritt 2:** Archiv extrahieren und folgende Archive in den Klassenpfad der Applikation aufnehmen:

```
antlr.jar
cglib-full.jar
asm.jar
asm-attrs.jars
commons-collections.jar
commons-logging.jar
ehcache.jar
hibernate3.jar
jta.jar
dom4j.jar
log4j.jar
```

Ein Beispiel …

Zuerst müssen alle Einstellungen vorgenommen werden, die global für die Benutzung einer Datenbank gelten. Dazu gehören beispielsweise die Connection-URL, die zu verwendende Implementation des Connection-Pools, der zu verwendende Transaktionsmechanismus, usw. Diese Einstellungen können innerhalb der Applikation, oder durch spezielle Konfigurationsdateien vorgenommen werden. Eine Möglichkeit ist die Angabe einer Java-Properties Datei mit Namen `hibernate.properties`, die im Klassenpfad der Applikation gespeichert, automatisch bei der Erzeugung einer Session eingelesen wird.

Glücklicherweise liefert die Hibernate Distribution eine Beispielkonfiguration `hibernate.properties` mit, die die meistgebräuchlichen Einstellungen enthält. Diese müssen dann nur noch auf die eigenen Bedürfnisse angepasst werden. Um einen Eindruck zu gewinnen, wie diese Einstellungen aussehen, zeigt Abb. 8.15 einen Ausschnitt aus der Datei, wie sie im Falle der Anbindung einer lokalen MySQL-Datenbank aussehen könnte.

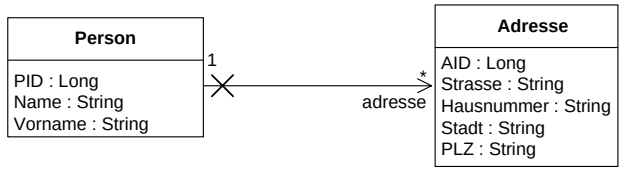
```
[…]
## MySQL

hibernate.dialect org.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class com.mysql.jdbc.Driver
hibernate.connection.url jdbc:mysql://localhost/test
hibernate.connection.username root
hibernate.connection.password
[…]
```

Abb. 8.15 Ausschnitt aus hibernate.properties

Haben wir alles korrekt konfiguriert, brauchen wir als nächstes Klassen, die wir persistieren können (Abb. 8.16).

Abb. 8.16 Beispielklassen




```
public class Person {

private Long PID;
private String name;
private String vorname;
private List adressen;

public Person {}

public void setPID(Long pid) {
      this.PID = pid;
}
public Long getPID() {
      return this.PID;
}

[...]

}
```

```
public class Adresse {

private Long AID;
private String strasse;
private String hausnummer;
private String stadt;
private String plz;

public Adresse {}

public void setAID(Long pid) {
      this.AID = pid;
}
public Long getAID() {
      return this.AID;
}

[...]

}
```

Als Trivialbeispiel dient uns hier die Klasse `Person` mit gerichteter *1-n-Assoziation* zu `Adresse` sowie die zugehörigen POJOs [POJO]. Hier wird ein weiterer Vorzug von Hibernate deutlich: Klassen, die unter Hibernate persistiert werden sollen, unterliegen keinen weiteren Forderungen an ihre Struktur (zu implementierende Interfaces, spezielle Basisklassen, …), als der, dass die zu persistierenden Attribute zugehörige getter-/setter-Paare besitzen sollten, wobei es sich selbst dabei um eine Kann-Forderung handelt, da Hibernate hier auch reflektiv arbeiten kann.

Bevor wir mit der Persistierung beginnen können, fehlt noch die Information, wie die Klassen auf ein vorhandenes Datenbank-Schema abgebildet werden sollen. Am besten geschieht dies durch die Angabe separater Mapping-Dateien, die im selben Verzeichnis wie die Klassen selbst abgelegt werden (Abb. 8.17, Abb. 8.18).

Abb. 8.17 Person.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-
3.0.dtd">

<hibernate-mapping>
    <class name="Person" table = "PERSON">
        <id type="long" column="PID">
```

```
            <generator class="increment"/>
        </id>
        <property name="name" column = "S_NAME"/>
        <property name="vorname" column = "G_NAME"/>
        <bag name="adressen" cascade="all">
              <key column="PID"/>
              <one-to-many class="Adresse"/>
        </bag>
    </class>
</hibernate-mapping>
```

Abb. 8.17 Person.hbm.xml (Forts.)

Die Bedeutung der einzelnen Einträge sind für unseren Zweck ausreichend intuitiv verständlich. Lediglich die Verwendung des `<bag>` Tags in `Person.hbm.xml` soll insofern näher erläutert werden, dass er die die *1-zu-n Beziehung* der Klassen untereinander realisiert. `<key column="PID" />` legt dabei fest, dass die Assoziation datenbankseitig als *Foreign-Key-Beziehung* der Spalte `PID` gelöst ist, der Parameter `cascade="all"` impliziert, dass eventuelle DELETE- und UPDATE-Funktionen auf tiefer gelegene Elemente des entsprechenden Objektbaums – in diesem Falle Objekte vom Typ Adressen – kaskadiert werden.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-
3.0.dtd">

<hibernate-mapping>

    <class name="Adresse" table = "ADDRESS">
        <id type="long" column="AID">
            <generator class="increment"/>
        </id>
        <property name="strasse" column = "STREET"/>
        <property name="hausnummer" column = "HNR"/>
        <property name="stadt" column = "CITY"/>
        <property name="plz" colum = "ACODE"/>
    </class>

</hibernate-mapping>
```

Abb. 8.18 Adresse. hbm.xml

Letztlich folgt noch die Beschreibung des Aufrufs aus dem Client (Abb. 8.19).

Abb. 8.19 Aufruf von Hibernate Funktionen aus einer Applikation

```
import net.sf.hibernate.*;
import net.sf.hibernate.cfg.*;

public class PersistenceTest {

private static SessionFactory sessionFactory;

public static void main(String[] argv) {

       try {
              // (1) get Hibernate configuration object
              Configuration cfg = new Configuration();

              // (2) classes to be controlled by Hibernate
              cfg.addClass(Person.class);
              cfg.addClass(Adresse.class);

              // (3) get session factory
              sessionFactory = cfg.buildSessionFactory();

              // (4) create new Person
              Person person = new Person();
              person.setName("Foo");
              person.setVorname("John");
                //create address
                Adresse adresse = new Adresse();
                adresse.setStrasse("Mullholland Drive");
                adresse.setCity("Los Angeles");

              // (5) add address to list
              person.setAdressen(new ArrayList());
              person.getAdressen.add(adresse);

              // (6) get Hibernate session
              Session session =
                      sessionFactory.openSession();

              // (7) save person
              session.save(person);

              // (8) close session
              session.flush();
              session.close();

       } catch (Exception e) {
              e.printStackTrace();
       }
   }
}
```

Literaturempfehlungen

Hibernate bietet eine solche Fülle von Konfigurations- und Anwendungsmöglichkeiten, dass selbst die Behauptung wir hätten das Thema angerissen schon fast eine Übertreibung darstellt. Wer also auf den Geschmack gekommen ist und sich näher über die vielfältigen Möglichkeiten informieren möchte, die ihm diese Technologie bietet, dem seien folgende Quellen empfohlen:

- Die *Homepage* des Projektes [Hibernate] bietet neben Forum, Mailinglisten, Dokumentation, APIs und FAQs reichlich Stoff zur Einführung.
- *Hibernate in Action* [BK04] von Bauer und King, bietet Anleitung von zweien der Hibernate Entwickler selbst und liefert damit Informationen aus erster Hand. Enthält viele anschauliche Beispiele.
- *Professional Hibernate* [PG04] von Pugh und Gradecki, ist ebenfalls kurzweilig zu lesen und enthält viele konkrete Code- und Konfigurations-Beispiele, die den Einstieg in die Arbeit mit Hibernate erleichtern.

8.2.3 Graphical Editing Framework (GEF)

Das Eclipse Graphical Editing Framework (GEF) aus dem Eclipse Tools Project stellt ein Rahmenwerk zur Entwicklung eines solchen Editors bereit und mindert damit den Entwicklungsaufwand erheblich. GEF setzt auf der Subkomponente Draw2D auf. Diese Komponente ermöglicht das Erstellen eigener, maßgeschneiderter, grafischer Komponenten.

Dem Anwender werden Basisfunktionalitäten für graphbasierte Editoren wie Knoten und Kanten, Zoom, Resizing in der API mitgeliefert und können ohne großen Aufwand eingebunden werden. Das Architekturmuster, dessen sich GEF dabei bedient ist das bekannte Model-View-Controller (MVC) (siehe [POSA1]).

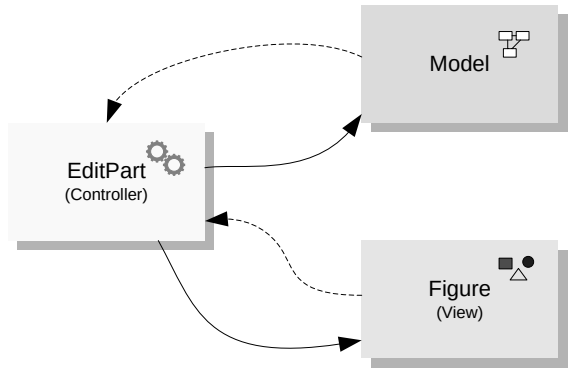
Die Abbildung von Model, View und Controller erfolgt dabei in GEF-Terminologie wie folgt:

- eine `Figure` entspricht dem *View* des Musters. Sie ist für die visuelle Repräsentation (die eigentliche Anzeige) zuständig. Sie muss dazu eine (abstrakte) Methode implementieren und sich darin selbst auf den `Canvas` *(dt. Leinwand)* zeichnen bzw. malen.

- Ein `EditPart` übernimmt die Aufgabe des *Controllers*. Abgeleitete Klassen müssen bzw. können verschiedene Methoden implementieren.
- Für die Rolle des *Models* gibt es hingegen keine Basisklasse, Modellklassen können daher beliebige Java-Klassen sein, sie müssen allerdings einen Mechanismus implementieren, mit dem sie den Controller über etwaige Zustandsänderungen benachrichtigen. Hier bietet sich die Umsetzung des Observer-Patterns an, damit das Modell unabhängig von View und Controller bleibt. Verwendet man *EMF* (ebenfalls in diesem Kapitel beschrieben) zur Definition des zugrunde liegenden Modells, so bekommt man diese Funktionalität geschenkt.

In GEF kennt ein View (=Figure) sein Modell nicht. Zudem fängt das Framework alle Benutzereingaben ab, indem es dazu passende Kommando-Objekte im Sinne des Command Patterns [GOF] erstellt, die direkt auf dem Modell arbeiten und so beispielsweise eine einfache Realisierung von Undo-Funktionen ermöglichen. Der Controller übernimmt also vollständig den anfallenden Datenfluss (siehe Abb. 8.20).

Abb. 8.20
MVC in GEF



Dies wird ermöglicht, indem man von abstrakten Klassen ableitet und diese implementiert was eine polymorphe Betrachtung ermöglicht. Für Knoten ist dies `AbstractGraphicalEditPart`, für Kanten `AbstractConnectionEditPart`.

Nach einer ersten steilen Lernkurve ist die Arbeit mit der GEF eine echte Erleichterung bei der Erstellung grafischer Editoren und „schenkt“ viele Funktionalitäten, deren Programmierung ohne die Hilfe des Frameworks nur absoluten Editor-Spezialisten vorbehalten

blieben. Das „Undo“-Feature wurde schon genannt, zusätzliche Beispiele sind:

- Automatisches Routing der verbindenden Kanten
- Layer-Funktionalitäten
- Reconnect von Kanten
- Automatische Rotation von Dekorationen (wie z. B. der Diamant bei Kompositionen) in Abhängigkeit der betroffenen Kante usw.

Literaturempfehlungen

Wie bereits erwähnt, ist der erste Einstieg in die Verwendung der GEF keine leichte Kost und kann daher an dieser Stelle nur so rudimentär geleistet werden wie im vorigen Abschnitt geschehen. Es existiert allerdings eine Fülle von Material im Internet und in Buchform, die diese Lücke schließt und auf die hier hingewiesen werden soll.

Erste Anlaufstelle sollte die *Homepage* des Projektes [GEF] sein, dort finden sich Links zu Online-Tutorials, die einen schnellen Einstieg ermöglichen.

Das Standard-Buchwerk in dieser Richtung ist [MDG+04], das nicht nur die GEF allein, sondern auch ausführlich das enge Zusammenspiel mit der EMF beschreibt. Letzte Details zur GUI-Entwicklung mit *SWT* und *JFace* können dann in [SHNM04] nachgeschlagen werden, das auch explizite Kapitel zu *Draw2D* und GEF enthält.

8.2.4 Eclipse Modeling Framework (EMF) + Eclipse UML2

Wollen wir mit Modellen arbeiten, ist es notwendig, die in Dateiform vorliegenden Modelldaten so aufzubereiten, dass wir softwaretechnischen Zugriff auf das Modell Repository (vgl. *UML- Repository* in Kapitel 3), und damit auf die Modell- bzw. Metamodellelemente erhalten.

EMF

Eine Schnittstelle auf eine solche Komponente, das die gewünschte Funktionalität bereitstellt, ist das *Eclipse Modeling Framework (EMF)* [EMF]. Wir werden EMF vor allem für die folgenden Aufgaben einsetzen:

- *Laden und Speichern* von Modellen (Deserialisierung/Serialisierung)
- *Aufbau des Modell-Repositorys* in Form eines Java-Objektbaums
- *Manipulation des Objektbaums* innerhalb der Durchführung von Transformationen, etc.

EMF ist als Plug-in konzipiert, welches sich nahtlos in Eclipse (siehe oben) integriert, sich aber auch „stand-alone" einsetzen lässt (siehe [EMFFAQ]). Es ist dem Eclipse Tools Project zugeordnet und wurde originär von IBM für die Entwicklungsumgebung *Visual Age* entwickelt. Das Ziel war es damals eine einheitliche Schnittstelle auf Metadaten zu liefern (vgl. [Fra05]). Im Jahre 2003 wurde EMF von IBM unter der Common Public License (siehe oben) in die Open Source gegeben.

Ursprünglich stützte sich die Implementierung des EMF auf die MOF 1.x. Erfahrungen bei der Übernahme in verschiedene Werkzeuge und deren spezielle Anforderungen machten jedoch Anpassungen des Modells notwendig und führten letztendlich zu einem eigenen (Meta-)metamodell genannt *Ecore*. Diese „optimierte Implementierung" zielte vornehmlich auf den Aspekt einfacher Werkzeugintegration und folgt weniger der primären Intention der MOF in erster Linie die Funktionalität eines Metadaten-Repository bereitzustellen (siehe [EMFFAQ]).

Mittlerweile ist IBM in den Spezifikationsprozess der MOF 2.0 involviert und forcierte dort die Trennung in EMOF und CMOF (siehe Kapitel 3 bzw. [EMFvsMDR]). Durch die Einflussnahme von IBM bzw. dem EMF-Team in die Spezifikation hat EMOF große Ähnlichkeit mit Ecore erhalten. In der aktuellen Version 2.0 kann Ecore bzw. EMF daher inzwischen im weiteren Sinne als Implementierung der EMOF angesehen werden, sowohl was die Abbildung auf XML bzw. XMI also auch auf Java betrifft. Nachdem man sich zwischenzeitlich von der MOF wegbewegt hatte, nähert man sich also nun wieder an. Unterschiede findet man allerdings beispielsweise in der Namensgebung der verwendeten Bezeichner. Diese ziert jeweils ein vorangestelltes „E", also etwa `EClass` statt `Class` (siehe unten).

Unter anderem ist dieser „Namenskonflikt" dafür verantwortlich, dass EMF nicht konform zu JMI [JMI] ist. Der von uns gewünschten Interoperabilität auf der Ebene des Datei-Austauschformats steht dies jedoch nicht weiter entgegen: EMF liest und schreibt XMI zwar

in der Standardeinstellung im eigenen Ecore-Format, kann dies aber auch im EMOF-Format bewerkstelligen und entspricht damit diesbezüglich dem Standard.

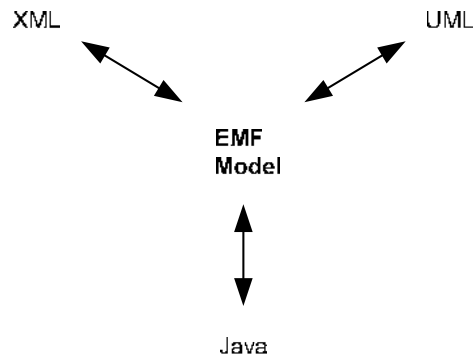


Abb. 8.21 EMF vereinheitlicht Java, XML und UML (Quelle: [BSM+03])

EMF wird in [BSM+03] als Vereinheitlichung von XML, UML und Java beschrieben (siehe Abb. 8.21). Hinter dieser Aussage verbergen sich im Wesentlichen die gerade gemachten Aussagen, also die Abbildung der MOF-Konzepte in die jeweiligen Konzepte der genannten Technologien ineinander.

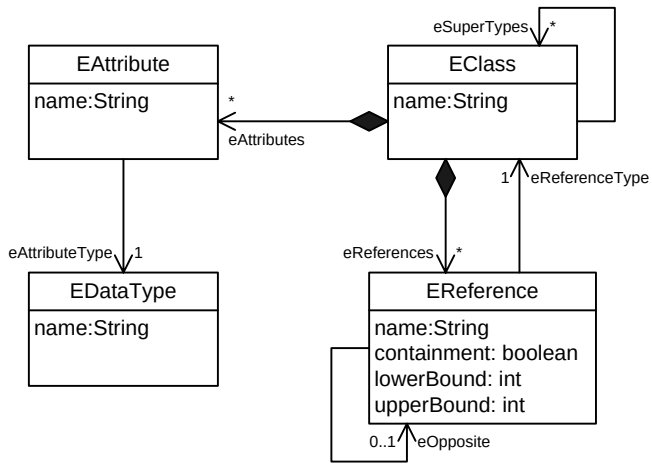


Abb. 8.22 Ecore Kernel

Ein Teil des EMF Metamodells – der *Ecore Kernel* – ist in Abb. 8.22 dargestellt. Die dort abgebildeten Konzepte Klasse (`EClass`), Attribut (`EAttribute`), Datentyp (`EDataType`) und Referenz (`EReference`) finden sich so oder in abgewandelter Form auch in

der EMOF wieder. Dort heißen sie `Class`, `Property` und `DataType`. Die Referenz wird dabei zusammen mit dem Attribut als `Property` subsumiert.

Wie bereits weiter oben angedeutet lassen sich die auf diesen Metakonzepten aufbauenden Modelle – die selbst als Metamodelle für Modelle dienen können – in drei Varianten verfassen:

- mit der UML, in Form eines *Rational Rose Models* (MDL-Format) [ROSE]
- mit *XML Schemata* [XS]
- mit annotierten Java-Klassen

Abb. 8.23 Von UML zu annotiertem Java

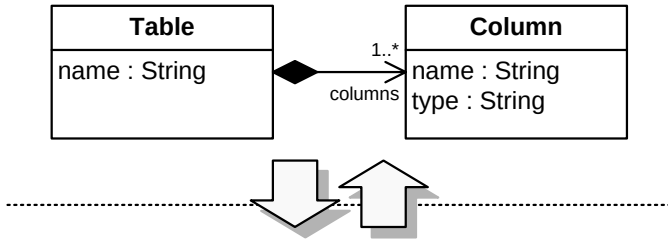


Table.java

```
Package dbexample;
import java.util.List;
/**
 * @model
 */
public interface Table {
  /**
   * @model type="Column"
   * containment="true"
   * required="true"
   */
  List getColumns();
  /**
   * @model
   */
  String getName();
}
```

Column.java

```
package dbexample;
/**
 * @model
 */
public interface Column {
  /**
   * @model
   */
  String getName();
  /**
   * @model
   */
  String getType();
}
```

Für die Teilmenge der angesprochenen Konzepte sind diese Formen äquivalent, sie lassen sich also ineinander überführen. Bei der Verwendung von annotierten Java-Klassen macht man dabei – wie der Name schon sagt – von Annotationen Gebrauch. Dies ist notwendig, weil sich mit den sprachlichen Mitteln von Java allein nicht alle Elemente abbilden lassen, etwa wie bei dem beispielhaft in Abb. 8.23 dargestellten *containment* der Assoziation zwischen `Table`

und `Column`. Wie dort zu sehen, sind die angesprochenen Annotationen als Javadoc Tags realisiert. Die Teilkomponente, die für diese Codegenerierung zuständig ist, heißt *Java Emitter Templates (JET).* Diese Technologie wird ausführlich in Kapitel 9 vorgestellt.

UML2

Als nächstes wollen wir uns dem Eclipse *UML2 Project* [EUML2a] zuwenden. Wie EMF ist es dem Eclipse Tools Project zugeordnet und unterliegt ebenfalls der CPL. Basierend auf der EMF-Technologie, die wie oben ausgeführt `MOF2::EMOF` implementiert, wurde in diesem Projekt das UML 2.x Metamodell umgesetzt. Abb. 8.24 illustriert diesen Zusammenhang noch einmal.

Um Missverständnisse zu vermeiden, werden wir im Folgenden die Schreibweise `UML2` für das aktuell betrachtete Framework verwenden, während UML weiterhin die Sprache UML meint.

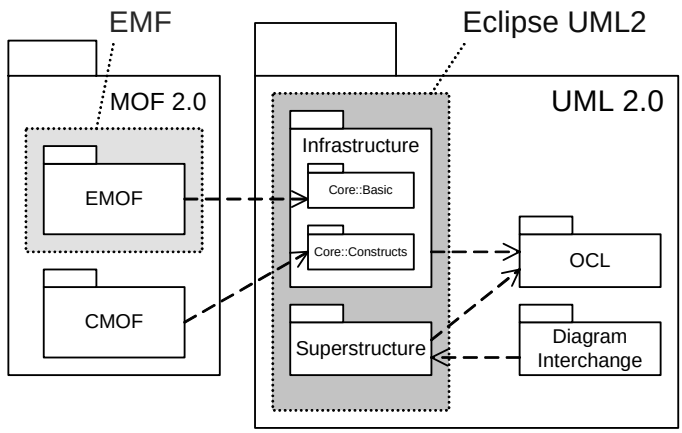


Abb. 8.24 EMF/Eclipse UML2 und MOF 2.0/UML 2.0

Im genua Framework wird `UML2` vor allem zur Erledigung der folgenden Anforderungen verwendet:

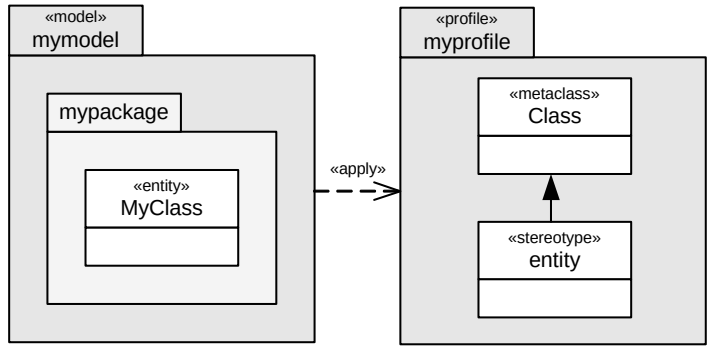
- dem Laden und Speichern von Modellen (Deserialisierung/Serialisierung),
- dem Aufbau des Modell-Repositorys, in dem sich alle Modelle befinden, die an Transformationen oder Codegenerierung teilnehmen
- und der Manipulation der Modellelemente innerhalb des Repositorys innerhalb von Transformationen.

Im Framework realisiert UML2 also die Funktionen des in Kapitel 3 beschriebenen UML-Repositorys. Dazu enthält es Java-Typdefinitionen, die das UML-Metamodell abbilden. Wird ein Modell aus einer Datei geladen werden die den Modellelementen entsprechenden Typen instanziiert. So wird im Speicher ein Objektnetz aufgebaut, das dem UML-Repository entspricht. Die UML2-API bietet Methoden, die auf diesem Repository arbeiten und das Modell programmatisch verändern können.

Beispiel: UML2-Modelle erzeugen

Um ein Gefühl dafür zu bekommen, wie die Arbeit mit UML2 aussieht, wollen wir ein Beispielmodell programmatisch erzeugen und die Codestellen, die die Aufgaben erledigen näher erläutern (Abb. 8.25).

Abb. 8.25 Beispielmodell



Erzeugt werden soll das Modell mymodel, das ein Paket namens mypackage enthält. Hier wiederum befindet sich die Klasse MyClass, die mit dem Stereotypen entity markiert ist. Der Stereotyp stammt aus dem Profil myprofile und erweitert die Metaklasse Class.

Wir werden die Erledigung der selbst gestellten Aufgabe mittels UML2 in drei Schritten betrachten: zuerst die Erzeugung des Modells inklusive der Elemente bis zur Klasse MyClass. Danach betrachten wir die Erzeugung des Profils, das den Stereotypen entity enthält. Als letztes wird dann beschrieben, wie das erzeugte Profil auf das bestehende Modell und der Stereotyp auf die Klasse angewendet werden kann.

Schauen wir uns nun den Code an, der den ersten Teil der Aufgabe erledigt (Abb. 8.26). Um mit der UML2-API arbeiten zu können,

müssen wir zunächst einige Initialisierungen vornehmen (1), die im Abschnitt *UML2-Helper* näher erläutert sind. Danach kann mithilfe der `UML2Factory` das Modell in Form der Klasse `Model` instanziiert (2) und der Instanz innerhalb des Repositorys der gewünschte Name verpasst werden (3).

```
/* Initialisierung */                                          (1)
init();
/* Modell erzeugen */
Model model = UML2Factory.eINSTANCE.createModel();            (2)
model.setName("mymodel");                                     (3)
/* Paket erzeugen */
Package package_ =
    model.createNestedPackage("mypackage");                   (4)
/* Klasse erzeugen */
package_.createOwnedClass("MyClass", false);)                 (5)
```

Abb. 8.26 Code-Snippets – Teil I

Als nächstes wird das Paketelement erzeugt. Die Hilfsoperation `createNestedPackage(…)` übernimmt diese Aufgabe und fügt das Element gleichzeitig dem Modell hinzu (4). Als Parameter wird ihr der gewünschte Name des Paketes mitgegeben, sodass auch dieser Schritt schon erledigt ist. Fehlt für den ersten Teil der Aufgabe nur noch die Klasse `MyClass` im eben erzeugten Paket. `createOwnedClass(…)` erledigt dies prompt (5). Auch ihr werden Parameter mitgegeben: der Name der Klasse sowie `false`, ein boolescher Wert, der hier anzeigt, dass es sich *nicht* um eine abstrakte Klasse handelt, was unserer Vorgabe entspricht.

Beispiel: UML2-Profile erzeugen

Nachdem wir die ersten Modellelemente erfolgreich erzeugt haben, wollen wir nun mit `UML2` das benötigte Profil erstellen (Abb. 8.27).

```
/* Profil erzeugen */
Profile profile = UML2Factory.eINSTANCE.createProfile(); (1)
profile.setName("myprofile");                             (2)
/* Stereotyp erzeugen */
Stereotype stereo =                                       (3)
    profile.createOwnedStereotype("entity", false);
/* Metamodell laden */
Model uml2Meta = (Model)                                  (4)
    load(URI.createURI(UML2Resource.UML2_METAMODEL_URI));
/* Metaklasse referenzieren */
org.eclipse.uml2.Class metaClass =                        (5)
```

Abb. 8.27 Code-Snippets – Teil II

Abb. 8.27 Code-Snippets – Teil II (Forts.)

```
    (org.eclipse.uml2.Class)
    uml2Meta.getOwnedType(
        UML2Package.eINSTANCE.getClass_().getName()
    );
profile.referenceMetaclass(metaClass);                    (6)
/* Extension erzeugen */
Extension extension =                                     (7)
    stereo.createExtension(metaClass, false);
/* Profil definieren */
profile.define();                                         (8)
```

Die Schritte (1)-(3) erzeugen das Profil sowie den enthaltenen Stereotypen mit den jeweils gewünschten Namen. Das Vorgehen entspricht dabei genau dem des vorigen Abschnitts, weshalb wir ihn nicht näher erläutern müssen. Interessant wird es erst wieder ab Schritt (4). Hier wird die Metaklasse referenziert, die der erstellte Stereotyp erweitern soll. Da der Stereotyp auf Klassen angewendet werden soll (vgl. Abb. 8.25) ist diese Metaklasse das Element namens `Class` aus dem UML2-Metamodell. Das Metamodell wird in (4) geladen (vgl. Abschnitt *UML2-Helper*) und in (5) das betreffende Element aus dem Repository „herausgefischt“. (6) Schließlich macht `Class` als Metaklasse in unserem Profil zur weiteren Verwendung verfügbar.

Als nächstes müssen wir die Verbindung des Stereotypen zu seiner Metaklasse in Form einer Extension herstellen (vgl. hierzu die Ausführungen zu Stereotypen in Kapitel 3). Schritt (7) erledigt dies mithilfe der Operation `createExtension(…)`. Die übergebenen Parameter sind die Referenz auf die betroffene Metaklasse und ein boolescher Wert, der anzeigt, ob die Verwendung des Stereotypen innerhalb eines Modells `required` ist oder nicht. Auch auf die Bedeutung dieses Wertes wurde in Kapitel 3 bereits eingegangen. Um erzeugte Profile verwenden zu können, müssen sie finalisiert werden, was in (8) auch passiert.

Beispiel: UML2-Profile anwenden

Wir haben nun das Modell und das Profil mit allen benötigten Elementen. Schließen wir nun den Kreis und bringen beide zusammen (Abb. 8.28). (1) wendet das Profil auf unser Modell an. Nun können wir die eben angelegte Klasse aus dem Modell heraussuchen (2) und in (3) schließlich unseren Stereotypen auf die Klasse anwenden: *Aufgabe erledigt.*

```
/* Profil anwenden */
model.apply(profile);                                          (1)
/* Stereotyp anwenden */
org.eclipse.uml2.Class clazz =                                 (2)
    (org.eclipse.uml2.Class)
        ((Package)model.getOwnedMember("mypackage")).
            getOwnedMember("MyClass");
clazz.apply(stereo);                                           (3)
```

Abb. 8.28 Code-Snippets – Teil III

Beispiel: UML2-Helper

Abschließend erörter wir der Vollständigkeit halber noch die Snippets, die fehlen, um das oben angeführte Beispiel in Eclipse zum Laufen zu bringen. Wir wollen den Code jedoch an dieser Stelle nicht erläutern und dies der im nächsten Abschnitt referenzierten Literatur überlassen.

```
private static void init() {

    String jarLocation =
        "jar:file:/<eclipsedir>"+              // ersetzen!
        "/plugins/org.eclipse.uml2.resources_1.1.0.jar!/";
    URI uri = URI.createURI(jarLocation);

    Resource.Factory.Registry.INSTANCE.
        getExtensionToFactoryMap().put(
            UML2Resource.FILE_EXTENSION,
            UML2Resource.Factory.INSTANCE);

    URIConverter.URI_MAP.put(
        URI.createURI(UML2Resource.LIBRARIES_PATHMAP),
        uri.appendSegment("libraries").appendSegment("")
        );

    URIConverter.URI_MAP.put(
        URI.createURI(UML2Resource.METAMODELS_PATHMAP),
        uri.appendSegment("metamodels").appendSegment("")
        );

    URIConverter.URI_MAP.put(
        URI.createURI(UML2Resource.PROFILES_PATHMAP),
        uri.appendSegment("profiles").appendSegment("")
        );
}
```

Abb. 8.29 Code-Snippets - Teil IV

Abb. 8.30 Code Snippets – Teil V

```
public static Package load(URI uri) {

Resource resource =
    new ResourceSetImpl().getResource(uri, true);
return (Package)
    EcoreUtil.getObjectByType(
        resource.getContents(),
        UML2Package.eINSTANCE.getPackage()
    );
}
```

Literaturempfehlungen

Die Homepage der Projekte [EMF] [EUML2a] bildet den zentralen Einstiegspunkt für den Download der aktuellsten Plug-in Versionen sowie vielen weiteren Informationen zu geplanten Milestones, dem Fortschritt des Projektes usw. Das `UML2` *Getting Started Tutorial* [EUML2b] erläutert ausführlich die Erstellung von Modellen und liefert die Details, die an dieser Stelle aus Platzmangel weichen mussten. Es wird durch [EUML2c] erweitert, das ausführlich die Erzeugung und Verwendung von Profilen erläutert. Die JavaDocs der aktuellen Version 1.1.1 sind unter [EUML2d] zu finden. Für EMF bildet [MDG+04] das Standardwerk, das in der Online-Version auch kostenlos als Redbook bezogen werden kann. [EMFFAQ] versucht die übrig gebliebenen Fragen zu klären.

TIPP! Der heißeste Tipp zum Schluss: Unter [ENEWS] finden sich die Informationen zum Erreichen der `UML2`-/EMF-Newsgroups. Hier können Fragen gestellt werden, die teilweise von Kenn Hussey selbst (dem „Hirn“ hinter `UML2`) bzw. durch weitere IBM-Mitarbeiter beantwortet werden. Die Reaktionszeit ist dabei erstaunlich schnell und die Atmosphäre selbst auch bei Anfängern sehr geduldig. Bei der Entwicklung des Frameworks haben wir hier die Informationen gefunden, die sonst nirgendwo zu bekommen waren. Dies gilt im Übrigen auch für die restlichen dort zu findenden Newsgroups: unbedingt reinschauen.

8.2.5 JavaServer Faces

Wie spätestens seit unserem „Rundumblick“ bekannt ist, wollen wir eine Internet-Anwendung bauen. Dementsprechend benötigen wir eine Technologie mit der wir dies in möglichst effizienter Weise bewerkstelligen können. Wir haben uns für die *JavaServer Faces (JSF)* entschieden.

Um die Beweggründe für diese Entscheidung zu verstehen (und gleichzeitig die Grundlage für die später anstehende Technologieprojektion zu legen) müssen wir etwas weiter ausholen. Schauen wir uns dazu zunächst Abb. 8.31 an. Diese zeigt das grundsätzliche Schema des HTTP-Protokolls, dem Protokoll mit dem HTML-Seiten im World Wide Web übertragen werden.

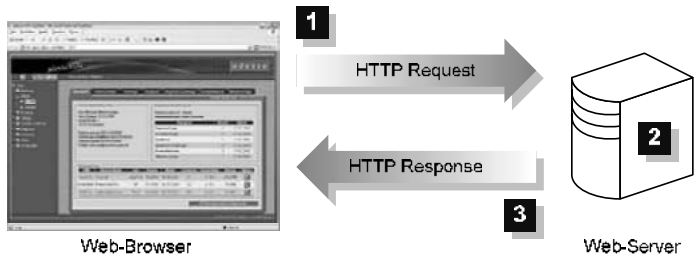


Abb. 8.31 Schema des HTTP Protokolls

Vereinfacht lässt sich dieses Schema wie folgt skizzieren:

1. Der Benutzer fordert über die Eingabe einer URL eine Web-Seite an. Der Browser setzt diesen Vorgang in ein Anfrage-Objekt (einen HTTP Request) um.
2. Dieses Anfrage-Objekt wird von einem Web-Server verarbeitet. Anhand der übertragenen Anfragedaten stellt dieser eine HTML-Seite zusammen.
3. Diese HTML-Seite wird in einem Antwort-Objekt (HTTP-Response) übertragen und im Browser dargestellt.

Für die Realisierung unserer Anwendung brauchen wir also ein Programmiermodell mit der wir die gerade schematisch beschriebene Funktionalität umsetzen können. Im Java-Umfeld sind dazu die Spezifikation der *Java Servlets* und der *JavaServer Pages* entstanden.

Die beiden Spezifikationen sind auf einem sehr techniknahen Abstraktionsniveau angesiedelt, wünschenswert wäre jedoch ein Programmiermodell, das es uns erlaubt eine Internet-Anwendung möglichst technologieneutral auf Basis von Dialogabläufen und Seitenbeschreibungen zu spezifizieren.

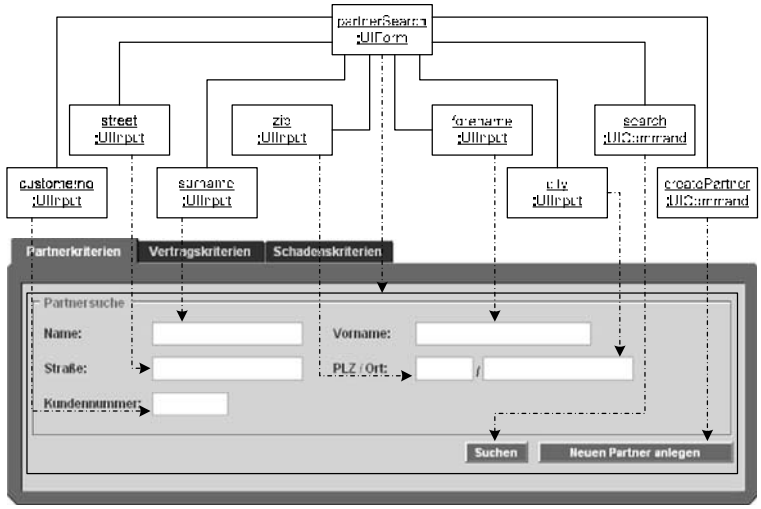
Web Application Frameworks

Zu diesem Zweck sind im Laufe der Zeit eine Reihe so genannter *Web Application Frameworks* entstanden. Dazu zählen etwa *Struts*, *Tapestry*, *WebWork* und *Spring*, um nur einige von ihnen zu nennen. Ein eigener Standard von Sun Microsystems zu diesem Thema fehlte lange Zeit.

Mit den *JavaServer Faces (JSF)* wurde diese Lücke 2004 geschlossen. Für die Spezifikation konnte man dabei Craig McClanahan als Specification Lead gewinnen [MBK04]. Er ist der Erfinder von Struts, das bis dato als *De-facto*-Standard im Bereich Web Application Frameworks galt.

Gegenüber anderen Web Application Frameworks zeichnet sich JSF dadurch aus, dass es beide oben genannten Aspekte bedient: Zum einen fußt es auf einem ereignisgesteuerten visuellen Komponentenmodell (siehe unten) sowie man dies bislang vornehmlich bei der Entwicklung von Rich Client Anwendungen (also etwa JFC/Swing [JFC]) antrifft, zum anderen bietet es die Möglichkeit der deklarativen Navigationsspezifikation.

Abb. 8.32 Visueller Komponentenbaum von JSF



Betrachten wir das visuelle Komponentenmodell. Abb. 8.32 zeigt dazu zunächst einen Ausschnitt einer HTML-Seite. Darauf zu sehen sind verschiedene visuelle Elemente, also etwa Eingabefelder und Schaltknöpfe. Diese Elemente haben in JSF eine *1:1* Repräsentation in Form einer Komponente. Komponenten können andere Komponenten aufnehmen – im Beispiel nimmt ein Formular die verschiedenen Eingabefelder und Schaltknöpfe auf. Das resultierende Geflecht von Komponenten hat eine baumartige Struktur, daher wird er visueller Komponentenbaum genannt.

Dieser Komponentenbaum wird anhand einer Seitenbeschreibung in Form einer JSP-Seite erzeugt. Eine solche Seite nimmt die Rolle des View innerhalb des von JSF implementierten MVC-Patterns [POSA1] an. Die Verknüpfung des Views zum Model erfolgt über eine Konfigurationsdatei (`faces-config.xml`). Aus dieser Konfiguration leitet sich auch das Verhalten des Controllers ab, der in einem Servlet realisiert ist.

Abb. 8.33 Zusammenspiel von Model und View in JSF

Betrachten wir dazu Abb. 8.33: der Wert (`value`) einer Eingabefeld-Komponente (`inputText`) zeigt auf das Attribut eines Modellelements (in JSF heißen diese Managed-Beans). Der Datenfluss von Model zu View und umgekehrt wird von JSF automatisch anhand dieser Informationen übernommen und erfolgt transparent für den Entwickler, er muss sich nicht mehr um das Mapping von Request-/Response-Daten der HTTP Anfrage kümmern.

Abb. 8.34 JSF Navigationsregel

So viel zur Seitenbeschreibung. Wie erfolgt aber die Bestimmung der Dialog- bzw. Seitenfolge? Sie wird ebenfalls in der Konfigurationsdatei `faces-config.xml` festgehalten. In Abb. 8.34 ist dazu ein entsprechender Ausschnitt dieser Datei wiedergegeben, in der eine so genannte Navigationsregel zu sehen ist. Die Elemente einer solchen Regel sind:

- Die *Ausgangsseite* (`from-view-id`)
- Eine oder mehrere *Navigationsfälle* (`navigation-case`)

Ein *Navigationsfall* wird durch eine *Aktion* (`from-action`), deren *Ausgang* (`from-outcome`) und die anzuzeigende *Folgeseite* (`to-view-id`) beschrieben.

Dies soll uns zunächst zur skizzenhaften Beschreibung der JSF-Technologie genügen. In den folgenden Kapiteln werden noch weitere Code-Snippets auftauchen, die die Arbeit mit JSF zeigen. Für eine ausführliche Einführung in die Materie verweisen wir auf [Ber03] und [HM04].

8.2.6 Apache Beehive

Wie wir im vorangegangen Abschnitt gesehen haben, besitzt JSF zwar die deklarativen Mittel Seitenfolgen einer Webanwendung zu spezifizieren, der Schwerpunkt des Standards wurde aber eindeutig auf das visuelle Komponentenmodell gelegt, wodurch die Möglichkeiten zur Modellierung von Interaktionsflüssen leider deutlich gelitten haben. Um in der Terminologie des MVC-Patterns (vgl. voriger Absatz) zu bleiben ist JSF also stark auf die Rolle des View fokussiert und – zumindest im Vergleich zu vielen anderen Webframeworks – weniger auf die des Controllers. Diese Schieflage ist wohl auch für einige der „Kinderkrankheiten" verantwortlich die man zuweilen JSF zuschreibt (vgl. [Ber03] und [GHH04]).

JSF: Zu sehr seitenorientiert

So wird beispielsweise immer wieder der Punkt kritisiert, dass JSF zu sehr seiten- und zu wenig dialogorientiert sei. Unter einem Dialog ist in diesem Zusammenhang eine zusammengehörige Abfolge von Ansichten/Masken, Systemaktionen und Subdialogen zu verstehen (z. B. „Partnersuche), die eine modulare Einheit bilden, und so als „Komponente" in verschiedenen Kontexten wiederverwendbar wären. JSF-Navigationsregeln erlauben zwar die Spezifizierung des Zusammenhangs zweier aufeinander folgender Seiten, ein Dialog mit breiterem Kontext lässt sich jedoch nicht explizieren.

Wollen wir nun selbst tätig werden und über JSF einen Dialog nachbilden, wartet viel Arbeit auf uns, da beispielsweise JSF nur Seiten-, Anfrage- und Sitzungs-Scope aber keinen Dialog-Scope kennt, und wir uns deshalb explizit um das Handling des Dialogzustands kümmern müssten. Darüber hinaus fehlt etwa die Möglichkeit zwischen erstmaligem bzw. wiederholtem Aufruf einer Ansicht zu

Unterschieden, was insbesondere die Vorinitialisierung von Ansichten schwierig werden lässt. Selbst wenn man über Eingriffe in den JSF-Lebenszyklus auch diese Schwächen in Eigenleistung beseitigen könnte erscheint uns das Grund genug uns nach bestehenden Lösungen umzuschauen, die uns diese Aufgaben abnehmen.

Die Lösung: Apache Beehive

Fündig wird man bei dem frisch aus dem Apache-Inkubator entschlüpften *Beehive* [Beehive]. Obwohl mit weit mehr Funktionalitäten ausgestattet, ist für unsere Zwecke vor allem das Webframework mit dem Namen *NetUI*, das neben einer JSP-Taglib für die View-Erstellung die Komponente *Page Flow* umfasst. Page Flows adressieren die von uns zuvor geschilderten Probleme und bieten gleichzeitig eine nahtlose Integration mit JSF, was es für unsere Zwecke ideal verwendbar macht.

Basiskonzepte

Beehive Java PageFlows (JPF) stützen sich im Wesentlichen auf drei Basiskonzepte ab, die wir im Folgenden näher betrachten wollen:

- *Logical Flow*: Anstatt direkt auf Folgeseiten zu verweisen, enthalten die JSPs eines PageFlows lediglich Verweise auf die Actions, die bei Auftreten bestimmter Ereignisse aufgerufen werden sollen, und dann darüber entscheiden, welche Seite als nächstes besucht wird. Auf diese Weise wird die Entscheidungslogik sauber von den darstellenden Komponenten getrennt.
- *Actions*: stellen eben jene Java-Methoden dar, die bei Auftreten einer Benutzer-Aktion aufgerufen werden (Betätigen eines Buttons, Auswahl einer Schaltfläche, usw.), und so die Ausführung von Geschäftslogik an Schlüsselstellen der Interaktion sowie die Ermittlung des nächsten Schrittes im Dialogfluss ermöglichen,
- *Controller-Klassen*: sind Java-Klassen die die Realisierung des MVC-Controllers darstellen und eine zusammengehörige Menge von Actions kapseln. Sie stellen damit das Werkzeug zur Modularisierung und Wiederverwendung sowie zur Verwaltung von Dialogflüssen zur Verfügung, das wir bei JSF vermisst haben. Dadurch, dass sie zustandsbehaftet sind, wird auch die Frage nach einem Session-Management auf Dialogebene sauber gelöst.
- Weiterhin ist die ausgiebige Verwendung von Metadata Annotations [JSR175], zur deklarativen Kennzeichnung der Control-

ler und Actions und vieler weiterer JPF-Elemente als Bestandteil zu nennen.

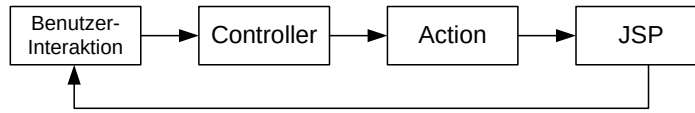


Abb. 8.35 Basisablauf einer JPF-Applikation

Schauen wir uns zur Verdeutlichung des generellen Ablaufs einer JPF-Webapplikation Abb. 8.35 an. Spezifizierte `Benutzerinteraktionen` resultieren im Aufruf des `Controllers`, der die Anfrage an die zugehörige `Action` weiterleitet. Hier erfolgt der Aufruf der Services aus der Geschäftslogikschicht der Applikation und schließlich die Entscheidung welche Seite (`JSP`) im Dialogfluss als nächstes aufgerufen wird.

Schauen wir uns nun einmal an, wie die bisher erwähnten Elemente in ihrer Java-Repräsentation ausschauen.

JPF-Controller

Abb. 8.36 Ausschnitt Controller-Klasse

```
import org.apache.beehive.netui.pageflow.PageFlowController;
import org.apache.beehive.netui.pageflow.annotations.Jpf;

@Jpf.Controller(                                              (1)
[…]
)

public class Controller
    extends PageFlowController                                (2)
{
    protected void onCreate()                                 (3)
    {
    }

    protected void onDestroy(HttpSession session)             (4)
    {
    }
}
```

Abb. 8.36 zeigt einen Ausschnitt aus einer Java Controller-Klasse. JPF-Controller sind durch die Annotation `@Jpf.Controller` gekennzeichnet (1) und erben von der Basisklasse `PageFlowController` (2) Ebenfalls dargestellt ist die Möglichkeit auf spezielle Ereignisse im Lebenszyklus des Controllers durch die Implementierung von Callback-Methoden zu reagieren. Hier kann beispielsweise

Code zur Initialisierung am Anfang (3), oder zum „Aufräumen“ am Ende eines Dialogflusses (4) hinterlegt werden.

Actions

JPF unterscheidet zwei Arten von Actions: `SimpleActions` und `ActionMethods`.

```
@Jpf.Controller(                                                    (1)
   simpleActions={
      @Jpf.SimpleAction(name=”logIn”, path=”login.jsp”)             (2)
   }
}
```

Abb. 8.37 Beispiel SimpleAction

SimpleActions werden auf Controller-Ebene definiert und kommen dann zur Anwendung, wenn kein Aufruf von Geschäftslogik, sondern nur die Funktionalität reiner Weiterleitung, Validierung und Weiterreichen der Benutzereingaben benötigt wird. Abb. 8.37 zeigt ein Beispiel für eine solche Aktion.

SimpleActions

Wird von einer Benutzerinteraktion im Rahmen des logischen Flusses die Aktion `logIn` vom Controller angefordert, leitet dieser sofort auf die Seite `login.jsp` um, die dann vom Container angezeigt wird (2).

Sollen vor der Verzweigung auf eine Folgeseite noch Services der Geschäftslogikschicht aufgerufen werden, oder handelt es sich um eine bedingte Verzweigung, bei der erst entschieden werden muss, welche Seite als nächstes aufgerufen wird (z. B. LogIn-Seite bei nicht angemeldetem Benutzer), kommen *ActionMethods* zum Einsatz.

ActionMethods

```
@Jpf.Action(                                                        (1)
 forwards = {
  @Jpf.Forward(name="notLoggedIn", path="logInNow.jsp")             (2)
  @Jpf.Forward(name="proceed", path="secure.jsp")                   (3)
 }
)
public Forward checkLogin (UserData user) {                         (4)
 // Benutzerdaten gültig?
          […]                                                       (5)
 if (loggedIn) return new Forward("proceed")                        (6)
  else return new Forward("notLoggedIn");                           (7)
}
```

Abb. 8.38 Beispiel ActionMethod

Abb. 8.38 zeigt ein Beispiel für eine „full-featured“ Action-Methode. Action-Methoden sind öffentliche Methoden innerhalb der Controller-Klasse, die einen Rückgabewert vom Typ `Forward` zurückliefern, und mit `@Jpf.Action` (1) gekennzeichnet sind. Ein *Forward* verknüpft einen repräsentativen Namen mit einer JSP. Jede Action-Methode liefert am Ende ihrer Ausführung ein Objekt vom Typ `Forward` zurück, das die Folgeseite im Dialogfluss bestimmt. Aktionen können nur Forwards zurückliefern, die zuvor innerhalb ihrer Annotation definiert worden sind. Die Menge aller Forwards einer Action entspricht also der Gesamtheit der möglichen Seiten, die als Folgeseite im aktuellen Dialogfluss in Frage kommen. In unserem Beispiel sind dies das Forward `notLoggedIn` (2), das zum Zug kommt, wenn der Benutzer noch nicht eingeloggt ist und dies erst auf der zugehörigen Seite `logInNow.jsp` nachholen muss bzw. `proceed` (3), das dann zurückgegeben wird, wenn der Benutzer bereits eingeloggt ist und den sicheren Bereich `secure.jsp` betreten darf.

Die Aktion wird dann aufgerufen, wenn eine Benutzerinteraktion beim Controller ihre Ausführung anfordert. Der Name der Anforderung entspricht dabei dem Java-Methodennamen. Die Seite kann beim Aufruf Daten mitliefern (typischerweise Formulardaten), die dann der Methode als Argument vom Controller übergeben werden. In unserem Fall sind dies die Daten des Benutzers vom Typ `UserData` (4).

Nun wird überprüft, ob der Benutzer sich bereits eingeloggt hat (5) und basierend auf dem Ergebnis der Überprüfung die Folgeseite ermittelt. Ist das LogIn bereits erfolgt (6) wird das in (3) definierte Forward zurückgeliefert, was im Aufruf der `secure.jsp` resultiert. Muss der Benutzer diesen Schritt erst nachholen (7), kommt (2) zum Zuge und die Seite mit dem LogIn-Formular kommt zur Anzeige.

Kommunikation der JSF-Komponenten an JPF

Die technischen Voraussetzungen zur Kombination der beiden Technologien liefert das Beehive-Framework. Die Anpassungen der Konfiguration, die die Verwendung der JSF-Komponenten statt der Beehive-eigenen NetUI-Elemente in der Präsentationsschicht erfordern, werden ausführlich in [Fei06] beschrieben.

An dieser Stelle wollen wir ein Beispiel zeigen, dass die Kommunikation der beiden Frameworks verdeutlicht. Betrachten wir zuerst Abb. 8.39, in der die JSF Login-Seite der Anwendung dargestellt ist. (1) stellt den Namen und die Position der Seite im Webkontext dar. Ein Aufruf wie etwa `www.mda-buch.info/genua-cs/logInNow.faces` würde also in der Ausführung und Anzei-

ge der entsprechenden Seite resultieren. In (2) und (3) findet sich die Bindung der Eingabefelder für Name und Passwort eines Benutzers an die Variable `backing`, die zur Laufzeit vom Framework mit den vom Benutzer eingetragenen Werten gefüllt wird. `backing` stellt eine Instanz des Typs `FacesBackingBean` dar, das ein Kernkonzept der Kommunikation von JSF mit Beehive darstellt, und das wir im Folgenden noch näher anschauen wollen.

genuacs/logInNow.jsp **(1)**

```
<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
  <title>To enter the secure area: LogInNow!(tm)</title>
</head>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<f:view>
  <h:form>
    <h:outputText value="Username:" />
    <h:inputText value="#{backing.user.name}" /></br>            (2)
    <h:outputText value="Password:" />
    <h:inputText value="#{backing.user.password}" /></br>        (3)
    <h:commandButton value="LogIn" action="checkLogIn">
      <f:attribute name="submitFormBean" value="backing.user" /> (4)
    </h:commandButton>
  </h:form>
</f:view>
</html>
```

Abb. 8.39 JSP der Login-Seite

Vorerst soll uns die Information genügen, dass in `backing` anscheinend getter- und setter-Methoden für `user` (`getUser()`, …) definiert sind, die wir aufrufen können, und in `user` gleiches für die Werte `name` und `password` gilt (vgl. Abb. 8.40).

```
public class UserData {

    private String name;
    private String password;

    // getter / setter and other stuff
    […]
}
```

Abb. 8.40 Klasse UserData

Benutzeraktion, die an den Beehive Controller weitergeleitet wird, ist das Drücken des Buttons `LogIn` (4). Vom Controller wird dann die Action `checkLogin` aufgerufen, die als Parameter den Wert `user` übergeben bekommt.

Abb. 8.41 zeigt die *Backing Bean (BB)*, die zur Kommunikation JSF – PageFlow verwendet wird. Backing Beans sind fest mit ihrer zugehörigen JSF-Seite verknüpft. Es gilt:

Wird eine JSF-Seite aufgerufen schaut das Framework nach, ob eine BB desselben Namens und Paketes wie die der Seite existiert. Existiert die BB wird sie instanziiert und der JSF zur Bearbeitung überlassen. Bei Verlassen der Seite wird die BB zerstört. Die Korrespondenz von Bean zu Seite folgt dabei folgendem Muster:

```
genuacs/logInNow.jsp <-> <src>/genuacs/logInNow.java
```

usw.

Abb. 8.41 Backing Bean

```
package genuacs;                                                    (1)

@Jpf.FacesBacking                                                   (2)
public class logInNow
  extends FacesBackingBean                                          (3)
{
  private UserData user = new User();                               (4)
  // getter / setter and more stuff
[…]
}
```

Neben der eben beschriebenen Korrespondenzbedingung müssen BBs noch die folgenden Kriterien erfüllen, um innerhalb einer JSF verwendet werden können (vgl. Abb. 8.41). BBs besitzen die Annotation `@Jpf.FacesBacking` (2) und erweitern `FacesBackingBean` (3).

Der Ablauf unserer Beispielanwendung sähe damit wie folgt aus:

- Der Benutzer ruft die Seite …/`genuacs/logInNow.faces` auf, worauf die in Abb. 8.39 definierte Seite zur Anzeige kommt.
- Das Framework findet die Backing Bean `logInNow.java` (Abb. 8.41), instanziiert sie, und stellt sie der JSF-Seite unter dem Namen `backing` zur Verfügung.

- Der Benutzer trägt seinen Namen und sein Passwort in die dafür vorgesehenen Eingabefelder ein und drückt den Knopf `LogIn`.
- Das Framework trägt die Werte in die Variablen `user.name` und `user.password` der BB ein (Abb. 8.39 (2)+(3)).
- Nun wird die Kontrolle an den Beehive Controller übergeben. Es wird die Aktion `checkLogin` angefordert und dem Aufruf der Parameter `backing.user` vom Typ `UserData` mitgeliefert (Abb. 8.39 (4)).
- Der Controller ruft die korrespondierende Methode auf (Abb. 8.38 (4)) und übergibt den mitgelieferten Parameter. Die Methode entscheidet dann, ob das Login gültig ist oder nicht und leitet dann an die entsprechende Seite weiter (6) + (7).

Literaturempfehlungen

Mit dem eben erläuterten Beispiel wollen wir unseren kurzen Überflug über Beehive beenden, und weitere Erläuterungen dem Fallbeispiel sowie dem Leser selbst überlassen.

Hervorragende Tutorials und Referenzdokumente zu Beehive finden sich auf der Beehive-Homepage unter [Beehive]. Die Integration von Beehive und JavaServer Faces wird genauestens unter [Fei06] unter die Lupe genommen und erklärt. Für Lesetipps zu JSF verweisen wir auf die entsprechenden Angaben in Abschnitt 8.2.5.

8.2.7 jBPM

Mit den *Beehive PageFlows* (vgl. vorigen Absatz) haben wir bereits eine Technologie kennen gelernt, mit der wir die Dialognavigation bzw. den Kontrollfluss einer Client-/Benutzer-Interaktion steuern können. Im folgenden Abschnitt wollen wir einen weiteren Mechanismus zum Management der Flusssteuerung einer solchen Interaktion betrachten, der jedoch zur Realisierung von Aspekten einer anderen Granularitätsebene verwendet wird.

Langlebige vs. kurzlebige Client/Benutzer-Interaktionen

Der angesprochene Unterschied liegt in der „Lebensdauer“ einer Sitzung, die vom jeweiligen Mechanismus kontrolliert wird. Während wir die Pageflows zur Abbildung von Workflow-Schrittfolgen verwendet haben, die „an einem Stück“ durchgeführt und im Normalfall nicht unterbrochen werden, wollen wir nun Abfolgen innerhalb langlebiger Geschäftsabläufe betrachten, die vom Benutzer für einen längeren Zeitraum unterbrochen und zu späteren Zeitpunkten wieder aufgenommen werden können. Um das Konzept noch einmal an einem Beispiel zu verdeutlichen, vergleiche man den Vorgang einer Partnersuche mit der Bearbeitung eines Antrags auf Versicherungsschutz. Während der erstgenannte Anwendungsfall mit dem

Fund oder Nichtfund des gesuchten Partners abschließt, kann es sein, dass die Aufgabe der Antragsbearbeitung unterbrochen werden muss, weil nicht alle Daten vorliegen und im Extremfall erst nach einigen Wochen wieder aufgenommen werden kann. Natürlich kann auch die Partnersuche, z. B. durch ein Telefongespräch, kurzzeitig unterbrochen und danach weiter fortgesetzt werden. Schließt der Benutzer jedoch die Anwendung oder schaltet er seinen Rechner ab ohne den Vorgang beendet zu haben sind die bisher getätigten Eingaben in den Partnermasken im Regelfall verloren und der Anwendungsfall *Partner suchen* muss erneut ausgeführt werden. Im Gegensatz dazu wäre der Anwendungsfall *Antrag bearbeiten* zwischen zwei erledigten Arbeitsschritten unterbrechbar, d. h. die Teilergebnisse dieser Schritte würden, für den Benutzer transparent, persistiert und könnten – im Beispiel bei Eintreffen der benötigten Informationen – dort wieder aufgenommen werden.

Die beiden vorgestellten Mechanismen betrachten somit Aspekte, die orthogonal zueinander sind, und ergänzen sich ideal zur systemtechnischen Realisierung langlebiger Client-/Benutzer-Interaktionen. Ein Geschäftsworkflow (realisiert mit Mitteln der jBPM) als langlebige Komponente besteht dann aus der Aneinanderreihung einzelner Geschäftsanwendungsfälle, die ihres atomaren Charakters wegen ideal durch Pageflows abgebildet werden können.

JBoss jBPM

Mit *JBoss jBPM* [JBPM] steht ein Workflow Management System zur Verfügung, das die eben beschriebenen Aufgaben zur Verwaltung langlebiger Benutzersitzungen übernimmt. Zusätzlich stellt es Funktionen zur Verwaltung von Benutzeraufgaben zur Verfügung, ermöglicht die Definition und Kontrolle des Kontrollflusses dieser Aufgaben (den unterliegenden Prozess) sowie die Zuordnung der Aufgaben zu Rollen und konkreten Akteuren.

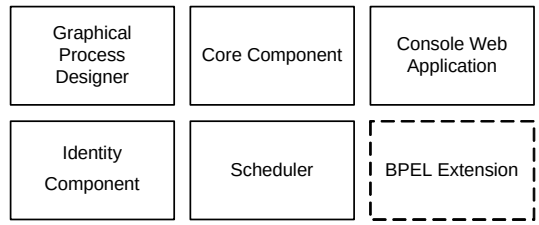


Abb. 8.42 Einige jBPM Komponenten im Überblick

Abb. 8.42 zeigt einen Überblick über die Bestandteile des Frameworks, von denen wir uns aufgrund des Platzmangels nur einige von ihnen im Folgenden näher anschauen können.

Oben links ist der *Prozessdesigner* zu erkennen, der eine grafische Modellierung von Geschäftsprozessen ermöglicht, und in Form eines Eclipse Plug-ins vorliegt. Daneben sieht man die *Core Component*, bei der es sich um eine Java SE API handelt, die Funktionen zum Management der Prozessdefinitionen in XML-Form sowie die Ausführung und das Monitoring von Prozessinstanzen zur Laufzeit ermöglicht. Mit der *Webkonsole* können Informationen über laufende Prozesse angezeigt und Parameter der Ausführung angepasst werden, während die *Identity Component* eine Verzeichniskomponente zum Management von Benutzer- und Unternehmensinformationen darstellt. Der *Scheduler* schließlich bietet Werkzeuge zur Überwachung und Ausführung von Timern und zeitgesteuerten Ereignissen in Prozessinstanzen und die *BPEL Extension* [BPELEXT], die nicht zum Basisumfang der Standarddistribution gehört (daher die gestrichelte Darstellung), ermöglicht die Verwendung von jBPM als BPEL Engine [BPEL].

Basiskonzepte

Kernkonzept der jBPM ist die *Prozessdefinition*. Sie stellt die formale Spezifikation eines Geschäftsprozesses dar und wird in XML-Form in der Datei `processdefinition.xml` vorgehalten. Prozessdefinitionen bestehen aus durch *Transitionen (engl. transitions)* verbundenen *Knoten (engl. nodes)*. Der Knotentyp spezifiziert das Verhalten des jeweiligen Knotens. Prozessdefinitionen können instanziiert und dann das Verhalten der so entstandenen Prozessinstanzen zur Laufzeit aus Java-Anwendungen heraus gesteuert werden. Die Ausführung beginnt immer am obligatorischen *Startknoten* und endet an einem *Endknoten*. Der Kontrollfluss „fließt“ entlang den Transitionen entlang von Knoten zu Knoten.

Token und Signale

Zur Bestimmung welcher bzw. welche Knoten als nächstes aktiviert werden dienen die Mechanismen *Token* und *Signal*. Ein Knoten wird dann aktiv, wenn ihn ein Token erreicht. Die Aussendung von Token wird durch Signale gesteuert. Diese Signale können beispielsweise direkt aus Java-Anwendungen heraus gesendet, aber auch durch den Ablauf von Zeitintervallen, dem Erreichen eines Zustandes oder Zeitpunkts, aber auch durch Ergebnisse von Entscheidungen, die an *Entscheidungsknoten (engl. decision nodes)* fallen, direkt von der Workflow-Engine getriggert werden. *Aktionen (engl. actions)* bilden einen weiteren Baustein der jBPM. Sie stellen benutzerspezifische Spezialisierungen von Knoten dar und repräsentieren Javacode, der bei speziellen Ereignissen in der Prozessausführung ausgeführt wird. Auf Aktionen gehen wir im weiteren Verlauf des Abschnittes noch genauer ein.

Erreicht die Ausführung einen Zustand, kann sie unterbrochen und die Prozessinstanz in einer Datenbank persistiert werden. Tritt ein Ereignis ein, das die Wiederaufnahme des Prozesses anstößt, wird die Prozessinstanz wieder aus der Datenbank geladen und die Ausführung geht weiter. Im Abschnitt über Code-Snippets werden wir sehen, wie dies mit Java-Mitteln bewerkstelligt wird.

Beispiel einer Prozessdefinition mit dem Eclipse Plug-in

Da handfeste Beispiele mehr sagen als tausend Erklärungen, schauen wir uns jedoch zuerst einmal an wie eine Prozessdefinition in jBPM, erstellt mit dem *Graphical Designer*, aussieht.

Installation

Da wie erwähnt der Designer nicht zur Standard-Distribution von jBPM gehört, muss er in Form eines Eclipse Plug-ins nachträglich installiert werden. Eine detaillierte Installationsanleitung findet sich unter [JBGD-GS], weshalb wir an dieser Stelle nicht näher auf Einzelheiten der recht einfachen Installationsprozedur eingehen wollen.

Mit dem mitgelieferten Wizard (File → New → Other → JBoss jBPM → Process Project) wird ein neues jBPM Projekt angelegt. Fügt man dann, ebenfalls unter Nutzung des Wizards, eine *Process Definition* namens `ex2` hinzu und klickt doppelt auf die so angelegte `processdefinition.xml` im Projektverzeichnis `src/process/ex2.par` öffnet sich der Editor des Designers (Abb. 8.43). Die dargestellte Abbildung zeigt den Prozess „Schenken-und-beschenkt-werden“, bei dem schon ein rudimentärer Ablauf modelliert wurde.

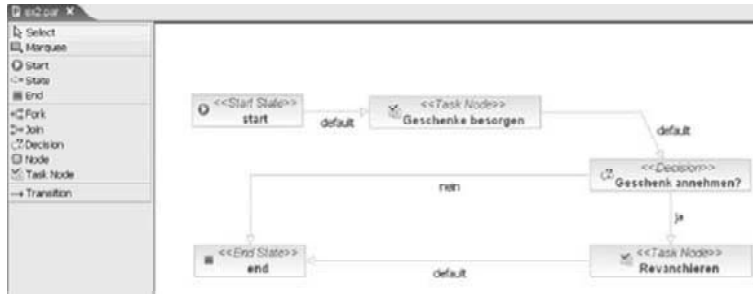


Abb. 8.43 Screenshot des Designer Plug-ins

Der Ablauf startet im *Startzustand (engl. start-state)* oben links und beginnt mit dem Aufgaben `Geschenke besorgen` zu müssen. *Aufgaben (engl. tasks)* können in jBPM unter *Aufgabenknoten (engl. task nodes)* zu Aufgabenpaketen gebündelt werden. So stellt der Knoten `Geschenke besorgen` nicht die Aufgabe selbst dar,

sondern repräsentiert lediglich eine Aufgabenmenge, deren Einzeltasks hier nicht weiter grafisch dargestellt sind. Ein Blick auf die *Outline* des Editors schafft hier weitere Einsichten (Abb. 8.44).

Eingekreist ist die Aufgabe `Geschenke besorgen` unter dem Aufgabenknoten gleichen Namens. Erreicht der Kontrollfluss den Aufgabenknoten, wird eine Aufgabe dieses Typs angelegt, deren Zustand (`erzeugt`/`zugeordnet`/`gestartet`/`beendet`) ab sofort an jeder weiteren Stelle des Prozesses abgefragt werden kann.

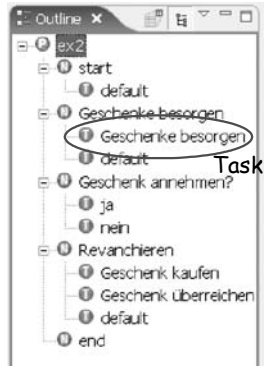


Abb. 8.44 Outline-Ansicht der Prozessdefinition

Aufgaben können optional Rollen oder Akteuren zugewiesen werden und werden dann Teil der Arbeitsliste des Akteurs (hier nicht verwendet).

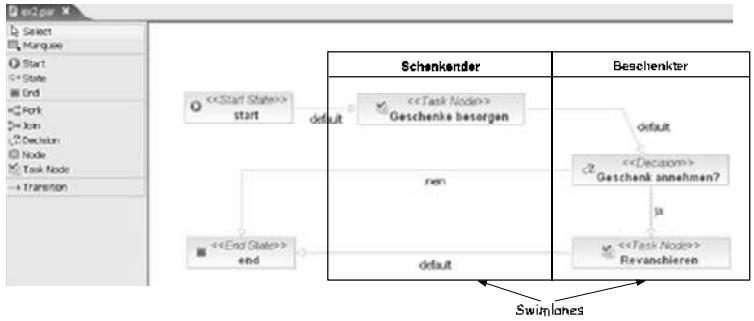


Abb. 8.45 Swimlanes

Standardmäßig wird erst dann mit der Ausführung des Prozesses fortgefahren, wenn alle Aufgaben eines Aufgabenknotens erledigt sind. Ist das Geschenk besorgt, die Aufgabe also als `beendet` markiert, kann es weitergehen. Der nächste Schritt im Ablauf besteht dann in der Entscheidung des Beschenkten, ob er das angebotene Präsent annehmen möchte oder nicht. Lehnt er ab, ist der Prozess beendet. Nimmt er an, ist er in der Pflicht sich zu revanchieren, was

gleich zwei Einträge in seiner Aufgabenliste erzeugt: `Geschenk kaufen` und `Geschenk überreichen` (siehe ebenfalls Abb. 8.44).

Wie aus der obigen Beschreibung des Prozesses hervorgeht, sind mindestens zwei Rollen an seiner Ausführung beteiligt: ein `Schenkender` und ein `Beschenkter`. Wie werden nun anstehende Aufgaben der richtigen Rolle zugewiesen? jBPM bietet dazu verschiedene Möglichkeiten. An dieser Stelle wollen wir nur die Verwendung von *Swimlanes* erläutern. Abb. 8.45 zeigt die Zuordnung der Aufgaben durch die grafische Darstellung des Swimlane-Konzeptes. Aktuell wird diese Darstellung nicht vom Designer unterstützt. Vielmehr wird eine Möglichkeit geboten Swimlanes zu definieren, und dann in der Outline-Sicht den verschiedenen Knoten zuzuweisen. Wie sich dies in der XML-Version der Definition niederschlägt, werden wir in Kürze sehen (Abb. 8.46).

Am Anfang (0) sehen wir das Tag `<process-definition>`, das die gesamte Prozessdefinition klammert. Darauf folgt der *Startknoten* (1) mit nur einer *Transition* (vgl. Abb. 8.45) namens `default`. Danach folgt die Definition der beteiligten Verantwortlichkeiten in Form von *Swimlanes* (2). Im Beispiel werden nur die Namen der beteiligten Rollen angegeben. jBPM bietet hier auch die Möglichkeit Java-Klassen anzugeben, die die *Zuordnung (engl. assignment)* der Aufgaben zu konkreten Akteuren zur Laufzeit übernehmen (hier nicht verwendet). Es folgt die Angabe der beteiligten Knoten (3) und abschließend als letztes Element die Angabe des Endzustandes (4).

Abb. 8.46 Ausschnitt process-definition.xml

```
[…]
<process-definition>                                    (0)
   <!-- START NODE -->                                  (1)
   <start-state name="start">
      <transition name="default"
                  to="Geschenke besorgen">
      </transition>
   </start-state>
   <!-- SWIMLANES -->                                   (2)
   <swimlane name="Schenkender"></swimlane>
   <swimlane name="Beschenkter"></swimlane>
   <!-- NODES -->                                       (3)
   <task-node name="Geschenke besorgen">
      <task name="Geschenke besorgen"
         swimlane="Schenkender">
      </task>
      <transition name="default"
                  to="Geschenk annehmen?">
      </transition>
   </task-node>
```

```
   <decision name="Geschenk annehmen?">
      <transition name="ja" to="Revanchieren">
      </transition>
      <transition name="nein" to="end">
      </transition>
   </decision>
   <task-node name="Revanchieren">
      <task name="Geschenk kaufen"
            swimlane="Beschenkter">
      </task>
      <task name="Geschenk überreichen"
            swimlane="Beschenkter">
      </task>
      <transition name="default" to="end">
      </transition>
   </task-node>
   <!-- END NODE -->                                    (4)
   <end-state name="end">
   </end-state>
</process-definition>
[…]
```

Abb. 8.46 Ausschnitt process-definition.xml (Forts.)

Im folgenden Abschnitt wollen wir einen kleinen Ausschnitt der verschiedenen Möglichkeiten betrachten, die uns die jBPM bietet, um die Art und das Verhalten von Knoten genauer zu spezifizieren sowie Elemente der Notation vorstellen, die wir bisher noch nicht kennen gelernt haben. Danach werden wir einen Blick darauf werfen, wie die praktische Arbeit mit jBPM innerhalb von Java aussieht.

Prozessmodellierung mit jBPM

Wir wissen bereits, dass eine Prozessdefinition in jBPM aus einer Menge von Knoten besteht, die durch Transitionen miteinander verbunden sind. Das Verhalten eines Knotens wird dann aktiviert, wenn ein Token den Knoten erreicht. Token werden durch Signale aktiviert, die entweder über die jBPM-API innerhalb einer Java-Anwendung getriggert werden können, aber auch an „externe“ Auslöser, wie dem Ablauf eines Zeitintervalls etc. gebunden sein können, die durch die Workflow-Engine selbst ausgelöst werden.

Was tatsächlich passiert, wenn ein Knoten aktiviert wird, ist vollständig durch seinen Typ bestimmt. Abb. 8.47 zeigt die möglichen Knotentypen.

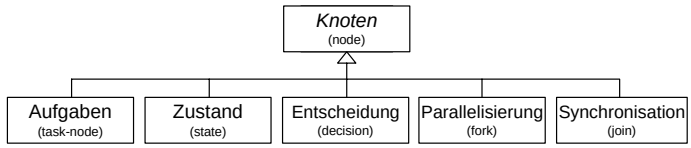


Abb. 8.47 Übersicht der möglichen Knotentypen

Als Basis dient der Typ `Knoten`, von dem alle weiteren Knotentypen erben. Spezialisierungen, die von jBPM selbst zur Verfügung gestellt werden und dem Benutzer damit sofort zur Modellierung von Prozessen zur Verfügung stehen sind:

- *Aufgabenknoten:* Diesen Knotentyp kennen wir bereits aus dem vorigen Beispiel. Hier können Aufgaben zusammengefasst werden, die dann optional in die Aufgabenlisten eines zugeordneten Akteurs hinzugefügt werden.
- *Zustände* repräsentieren reine Wartezustände, die weder Verhalten noch Einträge in Arbeitslisten hervorrufen. Sie dienen zum Anhalten des Prozesses bis zum Eintreffen eines erwarteten Signals (bspw. durch ein externes System o. Ä.), das dann den Kontrollfluss fortsetzt.
- *Entscheidungen*: Entscheidungsknoten werten *Bedingungen (engl. conditions)* aus, die an die ausgehenden Transitionen des Knotens angetragen sind. Zur Laufzeit wertet der Knoten alle Bedingungen der Reihe nach aus und setzt ein Token auf die Transition, deren Bedingung als Erste zu `true` ausgewertet wird. Entscheidungsknoten stellen eine von zwei Möglichkeiten zur Modellierung von bedingten Verzweigungen in der jBPM dar. Die zweite Möglichkeit besteht darin, die Entscheidung, welcher Weg als nächstes genommen werden soll extern zu treffen, und dann dem System im Signal zur Weiterreichung des Tokens mitzuteilen, welche Transition als nächstes schaltet.
- *Parallelisierungen* splitten ankommende Token und senden im Normalfall jeweils ein Token an alle ausgehenden Transitionen des Knotens. Mit diesem Vorgehen wird die Modellierung nebenläufigen Verhaltens ermöglicht.
- *Synchronisationen* stellen das Gegenstück zu Parallelisierungen dar. Sie werden erst dann aktiviert, wenn an allen eingehenden Transitionen Token anliegen und stellen damit so etwas wie ein UND innerhalb eines Kontrollflusses dar. Nebenläufige Kontrollflüsse, die durch den Einsatz von Parallelisierungen entstanden sind, können so wieder zusammengeführt werden.

Aktionen und Events

Bevor wir uns anschauen, wie die Verwendung von jBPM in Java aussieht und damit unseren kurze Vorstellung abschließen, wollen wir noch kurz auf die bereits am Anfang des Abschnittes erwähnten *Aktionen (engl. actions)* zu sprechen kommen. Sie stellen die Möglichkeit dar, beliebigen Javacode auszuführen, wenn definierte

Ereignisse (engl. events) im Ablauf des Prozesses eintreten. Abb. 8.48 zeigt die drei vordefinierten Ereignistypen, die jBPM standardmäßig liefert. Zusätzlich können beliebig viele Ereignisse selbst definiert werden.

Soll auf Ereignisse reagiert werden, müssen sie am betroffenen Prozesselement angetragen werden. Zusätzlich ist die Aktion anzugeben, die ausgeführt werden soll, falls das Ereignis den Knoten erreicht. Dies können vordefinierte Aktionen innerhalb der Engine sein, wie das Starten eines Timers, aber auch benutzerdefinierte Aktionen, die durch Spezialisierung der Klasse `ActionHandler` entstehen und deren Methode `execute(ExecutionContext ctx)` bei Auftreten des spezifizierten Ereignisses ausgeführt wird.

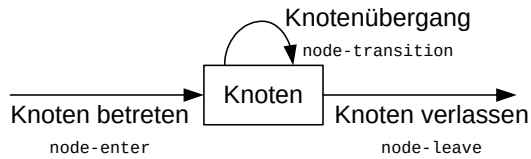


Abb. 8.48 Vordefinierte Ereignistypen

Die Variable `ctx` dient dabei als Kontext des Prozessablaufes, in dem übergreifende Prozessvariablen festgehalten werden können. Abb. 8.49 zeigt ein Beispiel für das eben Ausgeführte.

```
[…]
<state name="Arbeiten">                                    (1)
    <transition to="Nachhause laufen">                     (2)
        <action class="de.body.UpgradeFitness"/>           (3)
    </transition>
    <transition to="Nachhause fahren">                     (4)
        <action class="de.nomercy.BillDrivingCosts"/>      (5)
    </transition>
</state>
<state name="Nachhause laufen">
[…]
```

Abb. 8.49 Beispiel für Aktionen und Events

Maßgebliches Element für die betrachtungsrelevanten Ereignisse ist der Zustand `Arbeiten` (1), an dem zwei ausgehende Transitionen angetragen sind. Ist der nächste Zustand `Nachhause laufen` (2), wird die Aktion `UpgradeFitness` (3), getriggert durch ein `node-transition` Ereignis, ausgeführt. Im Falle, dass das Token den Weg zum Zustand `Nachhause fahren` nimmt (4), wird die `execute` Methode aus `BillDrivingCosts` ausgeführt (5).

Code-Snippets

Bevor wir diesen Abschnitt mit der Angabe von Literaturempfehlungen beenden und die weitere Beschäftigung mit jBPM auf die folgenden Kapitel verschieben, bzw. auf externe Literatur verweisen, werden wir uns noch kurz anschauen, wie die Einbindung von jBPM in Java-Anwendungen aussieht, indem wir mögliche Implementierungen einiger Schlüsselstellen im Lebenszyklus einer jBPM-Prozessinstanz vorstellen.

Prozessdefinitionen der jBPM liegen in Form einer Beschreibung vor, die mittels der *JPDL (jBPM Process Definition Language)* erstellt und in XML abgespeichert wird. Bevor eine solche Definition instanziiert und die so entstandene Prozessinstanz ausgeführt werden kann, muss die Definition zuerst in die Workflow Engine geladen werden. Abb. 8.50 zeigt eine Möglichkeit dies zu bewerkstelligen.

Mittels der statischen Methode `parseXMLReader(…)` aus `ProcessDefinition` (1) wird das XML-File gelesen und dann eine konkrete Prozessinstanz geschaffen (2).

Abb. 8.50 Prozessdefinition inital aus XML laden

```
ProcessDefinition definition =                                    (1)
    ProcessDefinition.parseXmlReader(
        new FileReader("/processdefinition.xml"));

ProcessInstance instance =
    new ProcessInstance(definition);                              (2)
```

Soll keine völlig neue Instanz geschaffen, sondern eine in einem Wartezustand unterbrochene Prozessausführung wieder aufgenommen werden, kann dies durch das Snippet der folgenden Abbildung (Abb. 8.51) geschehen. Da die Persistierung von Prozessinstanzen in jBPM mittels Hibernate bewerkstelligt wird, ähnelt der dazu notwendige Code den Snippets, die wir bereits in einem vorigen Abschnitt über diese Technologie kennen gelernt haben (vgl. Abschnitt 8.2.2).

Zuerst beschafft man sich eine jBPM-Session (1) und kennzeichnet dann den Beginn der Datenbank-Transaktion (2). Danach lädt man den persistierten Zustand der angegebenen Prozessausführung (3) und beschafft sich schließlich wieder eine Prozessinstanz, auf der wie gewohnt gearbeitet werden kann (4).

```
JbpmSession jbpmSession =
   jbpmSessionFatory.openJbpmSession();                    (1)

jbpmSession.beginTransaction();                            (2)

ProcessDefinition processDefinition =                      (3)
   jbpmSession.getGraphSession()
    .findLatestProcessDefinition("ex2");

ProcessInstance processInstance =                          (4)
          new ProcessInstance(processDefinition);
```

Abb. 8.51 Persistierten Prozess wiederaufnehmen

Signale aus Java-Anwendungen

Abb. 8.52 zeigt wie aus Java-Anwendungen Signale an die Prozessinstanz gesendet werden können, die den Tokenfluss auf dem Prozessgraphen beeinflussen. Zuerst liest man die aktuelle Position des Tokens auf dem Graphen aus (1). Danach wird das Signal zur Weiterreichung des Tokens an den nächsten Knoten des Graphen gegeben (2). Der Methode `signal(String transName)` wird dabei der Name der Transition als Parameter mitgegeben, den das Token nehmen soll – im Beispiel würde die Transition mit Namen `OK` gewählt. Fehlt der Parameter, wird die als Default definierte Transition genommen.

```
Token token = processInstance.getRootToken();              (1)
token.signal("OK");                                        (2)
```

Abb. 8.52 Token weiterreichen

Hat man die Arbeit auf der Prozessinstanz beendet, kann der aktuelle Zustand wieder in der Datenbank persistiert werden (Abb. 8.53). Zuerst beschafft man sich wieder eine Referenz auf die Session und speichert die Prozessinstanz durch Aufruf von `saveProcessInstance(...)` (1). Danach kann (in Fortführung des Beispiels aus Abb. 8.51) die Datenbank-Transaktion beendet (2) und die Session beendet werden (3). Dadurch wird die Unterbrechung des Gesamtprozesses zur späteren Wiederaufnahme (s. o.) ermöglicht und realisiert damit die Anforderung an die Unterstützung langlebiger Client/Benutzer-Interaktionen, die wir am Anfang des Kapitels als Kernanforderung erkannt haben.

```
jbpmSession                                                (1)
    .getGraphSession()
         .saveProcessInstance(processInstance);
jbpmSession.commitTransaction();                           (2)
jbpmSession.close();                                       (3)
```

Abb. 8.53 Prozessablauf persistieren

Literaturempfehlungen

Im Grunde verdient das jBPM Framework sein eigenes Buch, sodass an dieser Stelle nur die nötigsten Konzepte und Mechanismen vorgestellt werden können. Wir werden im Fallbeispiel noch einmal auf diese interessante Technologie zurückkommen, wer sich allerdings detailliert mit den Möglichkeiten befassen möchte, die jBPM dem Anwender bietet, der sei auf die folgenden Quellen im Internet verwiesen:

- Die Homepage des Projektes unter [JBPM], die neben einem Tutorial auch die vollständige Dokumentation des Frameworks und der JPDL bietet.
- Das Getting-Started des Graphical Designers [JBGD-GS], das in die Verwendung des grafischen Modellierungswerkzeuges unter Eclipse einführt.
- Sowie auf die URL zu diesem Buch *www.mda-buch.info* unter der zu allen hier vorgestellten Themen Informationen zu finden sind, die zur Drucklegung des Buches noch nicht verfügbar waren.

Literatur

[AL04] John Arthorne und Chris Laffra: *Official Eclipse 3.0 FAQs*. Addison-Wesley, 2004.

[Beehive] Apache Software Foundation (ASF): *Welcome to Apache Beehive*.
http://beehive.apache.org (letzter Abruf Januar 2006)

[Ber03] Hans Bergsten: *JavaServer Faces*. O'Reilly, 2003.

[BK04] Christian Bauer und Gavin King: *Hibernate in Action – Practical Object/Relational Mapping*. Manning, 2004.

[BPEL] International Business Machines Corporation (IBM) et al.: *Business Process Execution Language for Web Services v1.1*.
http://www-128.ibm.com/developerworks/library/specification/ws-bpel/ (letzter Abruf November 2005)

[BPELEXT] JBoss Inc.: *jBPM BPEL Extension*.
http://jira.jboss.com/jira/browse/BPEL (letzter Abruf November 2005)

[BSM+03] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick und Timothy J. Grose: *Eclipse Modeling Framework*. Addison-Wesley, 2003.

[Coc04] Alistair Cockburn: *Crystal Clear – A Human-Powered Methodology for Small Teams*. Addison-Wesley, 2004.

[Copyleft] Wikipedia: *Copyleft*.
http://en.wikipedia.org/wiki/Copyleft (letzter Abruf Mai 2005)

[CPL10] Eclipse.org: *Common Public License - v 1.0*.
http://www.eclipse.org/legal/cpl-v10.html (letzter Abruf Mai 2005)

[CPLFAQ] International Business Machines Corporation (IBM): *Common Public License (CPL) Frequently asked questions*.
http://www-106.ibm.com/developerworks/library/os-cplfaq.html (letzter Abruf Mai 2005)

[EMF] Eclipse.org: *EMF – Eclipse Modeling Framework*.
http://www.eclipse.org/emf (letzter Abruf Mai 2005)

[EMFFAQ] Eclipse.org: *Eclipse Modeling Framework FAQ*.
http://www.eclipse.org/emf/faq (letzter Abruf Mai 2005)

[EMFvsMDR] Martin Matula: *EMF vs MDR*.
http://mdr.netbeans.org/servlets/ReadMsg?list=users&msgNo=1877 (letzter Abruf Mai 2005)

[ENEWS] Eclipse.org: *Eclipse newsgroups home*.
http://www.eclipse.org/newsgroups/ (letzter Abruf November 2005)

[ETP] Eclipse.org: *Eclipse Tools Project*.
http://www.eclipse.org/tools (letzter Abruf Mai 2005)

[EUML2a] Eclipse.org: *The Eclipse UML2 Project*.
http://www.eclipse.org/uml2 (letzter Abruf Mai 2005)

[EUML2b] Kenn Hussey: *Getting Started with UML2*.
http://dev.eclipse.org/viewcvs/indextools.cgi/~checkout~/uml2-home/docs/articles/Getting_Started_with_UML2/article.html (letzter Abruf November 2005)

[EUML2c] Kenn Hussey: *Introduction to UML2 Profiles*
http://dev.eclipse.org/viewcvs/indextools.cgi/~checkout~/uml2-home/docs/articles/Introduction_to_UML2_Profiles/article.html (letzter Abruf November 2005)

[EUML2d] Eclipse.org: *Overview (Unified Modeling Language 2.0 Javadoc)*.
http://download.eclipse.org/tools/uml2/1.1.1/javadoc/ (letzter Abruf November 2005)

[Fei06] Rich Feit: *Integrating JavaServer Faces with Beehive PageFlow*.
http://dev2dev.bea.com/pub/a/2005/12/integrating-jsf-beehive.html (letzter Abruf Januar 2006)

[Fra05] David S. Frankel: *Eclipse and MDA – The Expanding Significance of the Eclipse Modeling Framework (EMF)*. In: MDA Journal – A BPT Column, Business Process Trends (www.bptrends.com), März 2005.
http://www.bptrends.com/publicationfiles/03-05 COL Eclipse and MDA - Frankel1.pdf (letzter Abruf Mai 2005)

[GB03] Erich Gamma und Kent Beck: *Contributing to Eclipse – Principles, Patterns and Plug-Ins*. Addison-Wesley, 2003.

[GBM03] David Gallardo, Ed Burnette und Robert McGovern: *Eclipse in Action – A Guide for Java Developers*. Manning, 2003.

[GEF] Eclipse.org: *Graphical Editing Framework*. http://www.eclipse.org/gef (letzter Abruf Mai 2005)

[GHH04] David Geary, Cay Horstmann und Cay S. Horstmann: *Core JavaServer Faces*. Prentice Hall, 2004.

[GOF] Erich Gamma, Richard Helm, Ralph E. Johnson und John Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Hibernate] JBoss Inc.: *Hibernate – Relational Persistence For Idiomatic Java*. http://www.hibernate.org/ (letzter Abruf November 2005)

[HM04] Sven Haiges (Hrsg.) und Marcel May: *JavaServer Faces - Web Development mit dem Standard-Framework*. Software & Support, 2004.

[JBGD-GS] JBoss Inc.: *JBoss jBPM Graphical Process Designer – Getting started*. http://docs.jboss.com/jbpm/v3/gpd/index.html (letzter Abruf November 2005)

[JBPM] JBoss Inc.: *JBoss jBPM*. http://www.jboss.com/products/jbpm (letzter Abruf November 2005)

[JFC] Sun Microsystems: *Java Foundation Classes (JFC/Swing)*. http://java.sun.com/products/jfc/ (letzter Abruf Mai 2005)

[JMI] Sun Microsystems: *Java Metadata Interface (JMI)*. http://java.sun.com/products/jmi/ (letzter Abruf Mai 2005)

[JSR175] The Java Community Process: *JSRs – Java Specification Requests – detail JSR# 175*.

[MBK04] Craig McClanahan, Ed Burns und Roger Kitain: *JavaServer Faces Specification – Version 1.1*. Sun Microsystems, 2004. http://java.sun.com/j2ee/javaserverfaces/download.html (letzter Abruf Mai 2005)

[MDG+04] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht und Philippe Vanderheyden: *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. International Business Machines Corporation, 2004.
http://www.redbooks.ibm.com/redbooks/pdfs/sg246302.pdf (letzter Abruf Mai 2005)

[OSI] Open Source Initiative (OSI): *Welcome*.
http://www.opensource.org (letzter Abruf Mai 2005)

[PG04] Eric Pugh und Joseph D. Gradecki: *Professional Hibernate*. John Wiley & Sons, 2004.

[POJO] Portland Pattern Repository's Wiki: *Plain Old Java Object*.
http://c2.com/cgi/wiki?PlainOldJavaObject (letzter Abruf Mai 2005)

[POSA1] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad und Michael Stal: *Pattern-Oriented Software Architecture – Volume 1 – A System of Patterns*. John Wiley & Sons, 1996.

[ROSE] International Business Machines Corporation (IBM): *Rational Rose XDE Developer – Family Overview*.
http://www-306.ibm.com/software/awdtools/developer/rosexde (letzter Abruf Mai 2005)

[SHNM04] Matthew Scarpino, Stephen Holder, Stanford Ng und Laurent Mihalkovic: *SWT/JFace in Action – GUI Design with Eclipse 3.0*. Manning, 2004.

[XS] World Wide Web Consortium (W3C): *XML Schema*.
http://www.w3.org/XML/Schema (letzter Abruf Mai 2005)

9 Projektdurchführung

„Wir bauen Software wie Kathedralen: zuerst bauen wir –
dann beten wir."

Gerhard Chroust

Auch das vorliegende Kapitel führt unser fiktives Projekt mit der Beschreibung der während des Projektverlaufes geschaffenen MDA-Infrastruktur (*genua*) weiter. Zur Illustration der Anwendung dieser Infrastruktur dienen Beispiele, die auf den Ergebnissen der Modellierung der Geschäftsprozesse auf berechnungsunabhängiger Basis (als CIM) aus der Analysephase aufbauen. Diese Beispiele werden zeitgleich mit der Beschreibung der technischen Framework-Komponenten eingeführt, um die Kernkonzepte besser erklären zu können. In der Realität würde zuerst das Framework erstellt werden müssen, bevor man die dort enthaltenen Assets verwenden könnte.

9.1 Architektur von genua Anwendungen

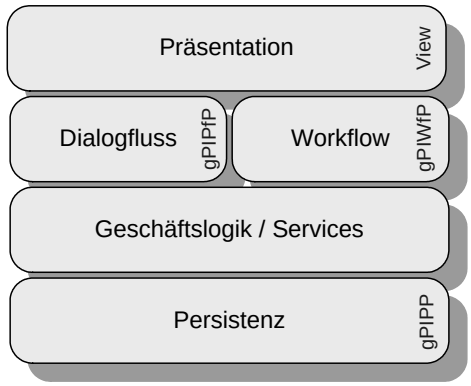
Bevor wir allerdings in die Tiefe und Details der einzelnen Umsetzungen gehen, wollen wir zuerst einen Überblick über die Gesamtarchitektur einer mittels *genua* erzeugten Anwendung geben. Dabei unterscheiden wir zwischen dem plattform-unabhängigen Design, das allen Anwendungen gemein ist, und den Realisierungsvarianten, die dieses konzeptionelle Gerüst in einen „harten" technischen Rahmen gießen.

Plattform-unabhängiges Design

Beginnen wir mit der plattform-unabhängigen Basisarchitektur, der alle *genua*-Anwendungen unterliegen. Abb. 9.1 zeigt, dass eine vierschichtige Architektur verwendet wurde. Zuoberst die *Präsentationsschicht* zur Darstellung der beteiligten Dialogmasken. Darunter

die Schicht, die sich um die *Kontrollflüsse* der Anwendung kümmert. Hier wird zwischen Flüssen auf Dialogsitzungebene und sitzungsübergreifender Workflow-Steuerung unterschieden.

Abb. 9.1 Architektur von genua-Anwendungen



Es folgt die Schicht, die den oberen Schichten die Geschäftslogik des Systems in Form von *Services* zur Verfügung stellt. Sollen im Rahmen dieser Geschäftslogik Daten persistiert werden, können sich die Services der Dienste der *Persistenzschicht* bedienen. Die berechneten Ergebnisse werden den darüber liegenden Schichten wieder verfügbar gemacht und gelangen letztendlich wieder dem Benutzer zur Anzeige.

Die beteiligten Profile und Metamodelle

Vertikal an die einzelnen Schichten sind die UML-Profile und (MOF-)Metamodelle angetragen, die zur Modellierung der Aspekte, für die sich die entsprechende Schicht verantwortlich zeichnet, verwendet werden können. Gemäß des plattform-unabhängigen Charakters dieser Architektur handelt es sich bei diesen ausnahmslos um Notationen zur Erstellung von PIMs. Im Einzelnen sind dies:

- *View*: Metamodell zur Modellierung von Dialogmasken und der Spezifikation der Interaktionsmöglichkeiten des Benutzers mit den dort enthaltenen Schaltflächen
- *genua Platform Independent Pageflow Profile (gPIPfP)*: Profil zur Modellierung von verzweigten Dialogflüssen
- *genua Platform Independent Workflow Profile (gPIWfP)*: Profil zur Definition von Workflows
- *genua Platform Independent Persistence Profile (gPIPP)*: last, not least ein Profil zur Modellierung des persistenten Charakters von Entitätsklassen

Die Bewertung der Tragfähigkeit und eventuelle Realisierung einer Möglichkeit zur plattfom-unabhängigen Modellierung von Geschäftslogik wurde auf eine spätere Phase des Projektes verschoben. Zu diesem Zeitpunkt werden nur Coderahmen generiert, die dann per Hand zur vollständigen Anwendung komplettiert werden müssen.

Technische Realisierungsvariante von M&M-online

Betrachten wir nun die technische Variante, die innerhalb des M&M-online Projektes gewählt wurde, um die plattformunabhängig spezifizierten Vorgaben zur realisieren (Abb. 9.2).

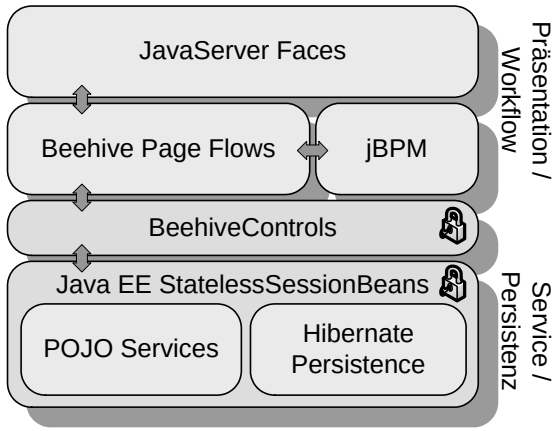


Abb. 9.2 Realisierungsvariante der Architektur

Auch hier handelt es sich um eine vierschichtige Architektur. Zur Realisierung wurden die folgenden Technologien verwendet:

- *JavaServer Faces (JSF)* zur Realisierung der Präsentationsschicht,
- *Beehive Page Flows* zur Abbildung von Dialogflüssen,
- *jBPM* zur Realisierung des Workflow-Aspektes
- sowie *Hibernate* zur Abbildung der Persistenz.

Alle oben aufgeführten Technologien erhalten während des Projektes ihr eigenes plattform-spezifisches Profil, um die PIM-zu-PSM Transformation und ein Feintuning vor der Generierung der binären Applikation zu ermöglichen.

Technische Invarianten

Obwohl die technische Realisierung eines Sachverhaltes, der mit dem plattform-unabhängigen Profilen des letzten Abschnitts modelliert wird, dem MDA-Prinzip folgend nicht vorgegeben wird, und es sich daher bei den gewählten Technologien lediglich um eine Möglichkeit handelt, die Vorgaben umzusetzen, existieren auch technische Invarianten, der alle Lösungsarchitekturen unterliegen. Sie sind in der ersten Version von genua obligatorisch und nicht variabel und stellen damit so etwas wie die Fixpunkte aller genua-Anwendungen dar.

Die betroffenen Schichten sind in obiger Abbildung ausgegraut und mit einem symbolischen Schloss versehen. So wurde im Rahmen des Projektes beschlossen, dass alle genua-Architekturen die Services der Geschäftsschicht in Java EE Stateless Session Beans kapseln und die Realisierung der Geschäftslogik „per Hand“ in Plain-Old-Java-Object Form (POJO) geschieht. Die Trennung dieser Services von den oberen, zugreifenden Schichten geschieht durch die Verwendung von Beehive Controls (vgl. Abschnitt 9.3.1). Dadurch ist es zwar prinzipiell möglich, die Implementierung der unteren Schichten durch andere Technologien auszutauschen, ohne Änderungen in den darüberliegenden Ebenen vorzunehmen, zu diesem Zeitpunkt wird jedoch von genua keine Unterstützung hierfür angeboten.

Nach diesen einleitenden Betrachtungen des Gesamtsystems wird im Folgenden der Verlauf des M&M-online Projektes detailliert sowie alle bisher vorgestellten Profile und Metamodelle entwickelt und am Beispiel erläutert. Auch für die gewählte technische Realisierungsvariante werden die benötigten Notationen angegeben und die Relation zu ihren plattform-unabhängigen Pendants erläutert, um als Basis zur Erstellung einer automatisierten Transformation/Generierung dienen zu können.

Im Rahmen des Themenkomplexes „Transformation und Generierung“ wird eine Sprache zur Spezifikation automatisiert ablauffähiger Transformationen namens gATL entwickelt und auch die Architektur und technische Einzelheiten der benötigten Tools – namentlich Transformator und Textgenerator – evaluiert und in Auszügen detailliert angegeben.

9.2 Dialoge und Kontrollflüsse

Bei der Betrachtung der genua-Architektur wollen wir von oben beginnen und uns dann schichtenweise nach unten durcharbeiten. Als

erstes steht daher der Themenkomplex „Modellierung von Kontrollflüssen und Masken“ an, der die ersten beiden Schichten der genua-Architektur – Präsentation und Kontrolflüsse – umfasst. Hier wurden im Kapitel Projektplanung (Kapitel 8) folgende Anforderungen als Ziel der ersten Iteration formuliert:

- die Erstellung eines Mechanismus zur plattform-unabhängigen Modellierung von Dialogflüssen und der in diesen Flüssen enthaltenen Dialogmasken,
- die Erstellung eines Profils zur Einbettung eines Dialogflusses in einen Workflow-Kontext,
- die Erzeugung von plattform-spezifischen Metamodellen/Profilen zur Darstellung der Sachverhalte auf technologie-spezifischem Abstraktionsniveau
- sowie letztendlich die Angabe der Elementrelationen zwischen den Darstellungen der unterschiedlichen Abstraktionsstufen zur Ermöglichung einer automatisierten Transformation in lauffähigen Quellcode.

Der folgende Abschnitt widmet sich der Modellierung von Dialogmasken und -flüssen. Danach werden die Mechanismen entwickelt, die zur Darstellung und Realisierung der Workflow-Aspekte notwendig sind.

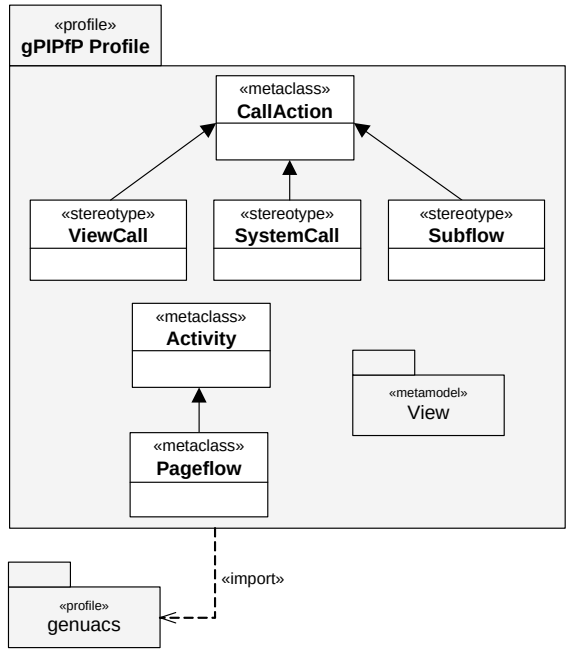
9.2.1 genua Platform Independent Pageflow Profile (gPIPfP)

Wie angekündigt beginnen wir mit der Modellierung von Dialogflüssen im Sinne von zusammenhängenden Maskenfolgen, die eine in sich geschlossene, wiederverwendbare Einheit bilden, und oft der Realisierung eines Geschäftsanwendungsfalls aus der Geschäftsprozessanalyse entsprechen (auf den Unterschied zwischen Dialogflüssen und Workflows wurde bereits in 8.2.6 bzw. 8.2.7 eingegangen).

Einen ersten Prototyp zur Modellierung eines Dialogflusses auf plattform-unabhängiger Ebene, stellt das *genua Platform Independent Pageflow Profile (kurz gPIPfP)* dar (Ja, der akronymartige Ansatz zur Kurzbeschreibung skaliert wahrscheinlich nicht ;o)). Das Profil ist baukastenartig aufgebaut und besteht aus drei Basismodulen, die wir im Folgenden einzeln betrachten und an einem Beispiel näher erläutern wollen.

Abb. 9.3 stellt die Definition des Profils in grafischer Form als UML-Diagramm dar.

Abb. 9.3 Definition von gPIPfP



Als Basisdiagrammart wurde das UML-Aktivitätsdiagramm gewählt und einige Diagrammelemente durch das Profil erweitert, um den gewünschten semantischen Umfang umsetzen zu können. Neben den auf oberster Ebene enthaltenen Elementen enthält das Profil noch weitere Konstrukte, die als Module eingebunden werden: zum einen das Metamodell `View` (Abb. 9.6), das Konstrukte zur Modellierung von Dialogmasken enthält sowie durch Import das Profil `genuacs` (Abb. 9.4), das Elemente enthält, die im weiteren Fallbeispiel noch öfter benötigt werden, und deshalb ebenfalls ausgelagert wurden. In `genuacs` wird unter anderem die UML-Superstructure importiert, was die Erweiterung des Aktivitätsdiagramms sowie die der weiteren UML-Basiselemente erst ermöglicht.

Basiskonzept

MVC-basiertes Design

Die Modellierung von Dialogflüssen in gPIPfP basiert auf den Prinzipien des MVC-Patterns (siehe etwa [POSA1]), wobei zur Erreichung einer möglichst großen Variationsbreite in den nachfolgenden plattform-spezifischen Ebenen des Modellzyklus für die Anbindung von `View` und `Model` eine Pull-Variante des Patterns gewählt wurde. Zur Kommunikation zwischen `Controller` und `View` werden

keine dezidierten Datencontainer (z. B. Formbeans, o. Ä.) verwendet (wie ein so genanntes Kontext-Mapping dennoch realisiert werden könnte wird in den Ausführungen weiter unten kurz angerissen). Views bekommen den Ausschnitt des Modells, der zur Anzeige ihrer `ViewElemente` benötigt wird, vom `Controller` geliefert, auf dem sie dann direkt arbeiten. Auf diese Weise werden sowohl Realisierungen der Views in Fat-Client Umgebungen (z. B. Swing, SWT), als auch web-basierte Ansätze ((dyn.)HTML) ermöglicht – die Fokussierung in genua liegt aber auf zuletzt genannter Variante.

Innerhalb des MVC-Kontextes besitzt jedes Element des Profils eine klar abgegrenzte Aufgabe:

- *Modellelemente* sind reine Datencontainer und enthalten keine View-spezifischen Anteile,
- *Views* konzentrieren sich auf die Repräsentation der Modelldaten und werden nicht mit Businesslogik vermischt.
- Der *Pageflow* verwaltet das Modell einer Sitzung (Pageflow-Context) und bestimmt den Kontrollfluss der Dialogfolge, indem er entscheidet, welches View oder welche Systemaktion als nächstes zur Ausführung gelangt.

Im Laufe des Abschnittes werden wir die Bedeutung der einzelnen Elemente des Profils und ihr Zusammenspiel noch näher kennen lernen.

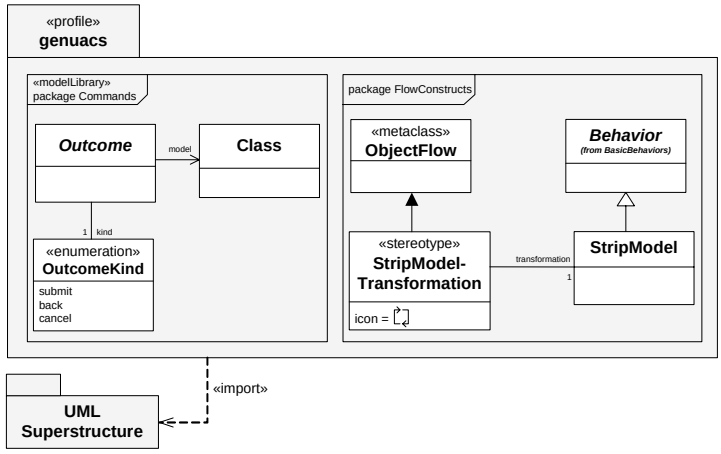


Abb. 9.4
genuacs Profil in UML-Darstellung

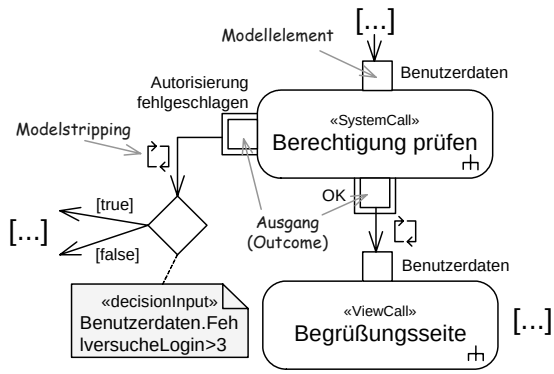
Die Grundlage der Modellierung von Dialogflüssen mittels gPIPfP basiert auf der Idee, diese als Abfolge des Aufrufs von Views

(«ViewCall») und optional zwischengeschalteter Aufrufe von Geschäftslogik bzw. Services («SystemCall») zu beschreiben, die als spezialisierte Aktionen realisiert sind. Den Kontrollfluss bestimmt die übergeordnete Aktivität in Form des «Pageflow» wahlweise auf Basis des ebenfalls vom Pageflow verwalteten, sitzungsbehafteten Modells (PageflowContext) oder aufgrund der Rückgabe der View-/ SystemCalls. Diese Rückgabe besteht im Normalfall aus einem *Ausgang(-styp) (engl. Outcome)* als Spezialisierung der abstrakten Oberklasse *Outcome*, die im Profil genuacs enthalten ist (siehe Abb. 9.4). Während Aufrufe von Views immer eine Spezialisierung von *Outcome* zurückgeben, können Systemaktionen sich wahlweise auch auf die Rückgabe „purer" Datenpakete beschränken.

Kontrollflüsse

Als Trivialbeispiel zur Erläuterung der Behandlung von Kontrollflüssen in gPIPfP gelte hier eine Systemaktion, die die Autorisierung des Nutzers für die Anzeige des nächsten Views überprüft (Abb. 9.5).

Abb. 9.5 Beispiel Kontrollfluss



Der angenommene Kontroll-/Objektfluss startet am oberen Rand der Abbildung. Die Systemaktion Berechtigung prüfen wird durch das Eintreffen der Benutzerdaten aktiviert und ruft das ihr zugeordnete Verhalten (ein Service der Geschäftslogik) mit diesen Daten als Eingabeparameter auf.

Der weitere Fluss der Dialogfolge ergibt sich nun aus dem Ergebnis des Serviceaufrufs. Hier bestehen zwei Möglichkeiten: der Autorisierungsversuch ist erfolgreich oder er schlägt fehl. Der Service teilt im vorliegenden Falle den Ausgang dadurch mit, dass er einen entsprechenden *Ausgang* erzeugt (hier OK oder Autorisierung fehlgeschlagen). An das Ausgangsobjekt kann über die

model-Relation (vgl. Abb. 9.4) ein Nutzdatenpaket mitgeliefert werden, das zur weiteren Auswertung (z. B. an einem Entscheidungsknoten) oder als Eingabe des nächsten Views bzw. einer weiteren Systemaktion verwendet werden kann. Dient das angehängte Datenpaket als Eingabe des nächsten Knotens (Systemaktion, Entscheidungsknoten, View) kann es mittels einer speziell stereotypisierten Kante vom Ausgangsobjekt getrennt werden. Hierdurch wird die Kompatibilität der Input-/Outputpins gewährleist jedoch gleichzeitig die Folgeaktionen nicht von speziellen Ausgängen abhängig gemacht.

Um eine solche Transformation handelt es sich bei dem Stereotypen «StripModel», der nichts anderes macht, als das Modell vom Ausgangsobjekt (Outcome) zu trennen und nur das abgetrennte Datenpaket im Objektfluss weiterzureichen. Anstatt des Stereotyps kann auch seine Repräsentation als Icon verwendet werden: ein stilisiertes Transformationssymbol in Quadratform.

Verzweigung von Kontrollflüssen durch ParameterSets

Die Entscheidung, welcher Zweig im Kontrollfluss als nächstes genommen wird, hängt in gPIPfP Diagrammen somit vorzugsweise vom zurückgelieferten Ausgang(-sobjekt) ab. Um diese Exklusiv-ODER-Verknüpfung von Aktionen zu realisieren, wurde das ParameterSet-Konstrukt der Aktivitätsdiagramme verwendet (vgl. Anhang A: UML-Schnellreferenz). Schlägt Berechtigung prüfen fehl, liefert die Systemaktion den Ausgang Autorisierung fehlgeschlagen zurück, an dem als model die Benutzerdaten angehängt sind. Die ausgehende Kante ist mit «StripModel» stereotypisiert, sodass der folgende Entscheidungsknoten das Ausgangsobjekt selbst nicht mehr „sieht“, sondern den Check, ob das Limit fehlgeschlagener Versuche überschritten worden ist, direkt auf Benutzerdaten ausführen kann. Auch im Falle der positiven Autorisierung wird ein Ausgangsobjekt (OK) als Output-Parameter zurückgegeben an dem die Benutzerdaten hängen. Der nachfolgende View verwendet diese Daten, um den Angemeldeten personalisiert begrüßen zu können. Durch die Verwendung von «StripModel» wird auch hier erreicht, dass die Folgeaktion lediglich vom benötigten Input (Benutzerdaten) und nicht vom konkreten Ausgang abhängig ist.

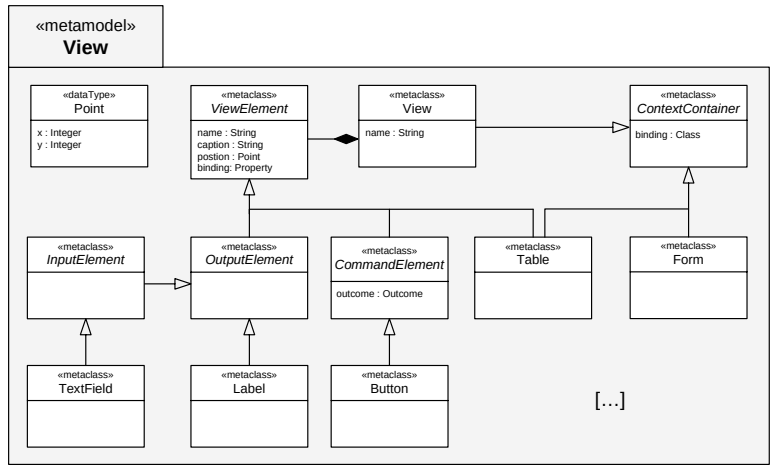
Viewmodellierung

Betrachten wir nun, wie die Views des Dialogflusses näher spezifiziert werden können. Hier sind zwei Aspekte interessant:

- die Definition der *Ein- und Ausgabeelemente* des Views, die sein Aussehen und die Interaktionsmöglichkeiten bestimmen sowie
- die *Zuordnung von Ausgängen (engl. Outcomes)* zu Benutzerinteraktionen und den davon betroffenen zu aktualisierenden Elementen des Pageflow-Kontextes.

Um die Modellierung dieser Aspekte zu ermöglichen, wurde innerhalb des Projektes das Metamodell `View` erstellt, das modularer Bestandteil des gPIPfP Profils ist (Abb. 9.6).

Abb. 9.6 View-Metamodell



Basiselement ist der `View`, dem eine beliebige Anzahl `ViewElemente` zugeordnet werden können, die das Aussehen der Dialogmaske bestimmen. Neben den `OutputElementen`, die die übergebenen Modelldaten in grafischer Form repräsentieren, und den `InputElementen`, die die angebundenen Werte des Pageflow-Kontextes durch Benutzereingaben aktualisieren, sind die *`CommandElemente`* zu nennen, an die ein Outcome angehängt werden kann. Wird ein solches Element vom Benutzer aktiviert (beim `Button` wäre das der Klick mit der Maus) liefert der zugeordnete `ViewCall` das assoziierte Ausgangsobjekt zurück und bestimmt so den weiteren Kontrollfluss. Welche Nutzdaten dabei „mitgeschickt" werden bestimmt ein Element des Typs *`ContextContainer`*. An dieses werden die `ViewElemente` angehangen, deren zugeordneten Datenelemente bei Auslösen eines Ausgangs durch eventuelle Benutzereingaben überschrieben bzw. aktualisiert werden sollen. Die Zuordnung von Augangs- zu View-/Datenelementen geschieht

implizit durch eine Assoziation zwischen Ausgangselement und Container.

Abb. 9.7 zeigt die Anwendung des View-Metamodells noch einmal am Beispiel.

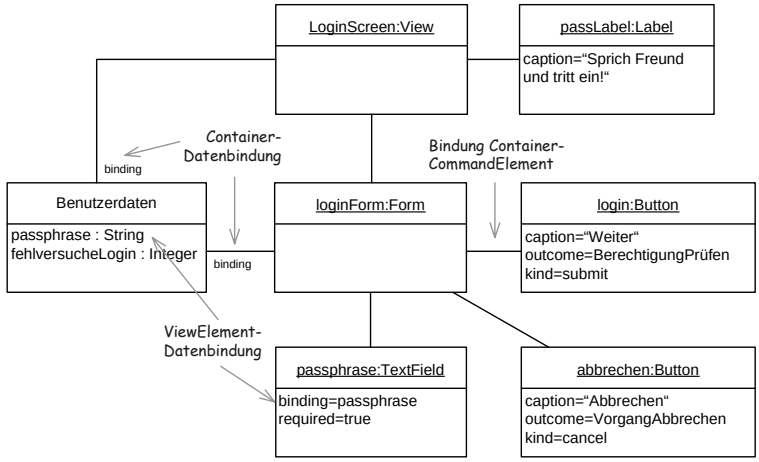


Abb. 9.7 Anwendung des View-Metamodells

Dargestellt ist die Modellierung des Dialogs `LoginScreen`, der dem Beispiel aus Abb. 9.5 vorangegangen sein könnte. Es handelt sich um eine einfache Eingabemaske, mit nur vier Elementen: einem Label, das zur Eingabe der geheimen Losung auffordert, einem Textfeld, das genau diese Eingabe ermöglicht, sowie zwei Knöpfen `login` und `abbrechen`, von denen der eine die gemachten Eingaben (das der `LoginForm` zugeordnete Datenelement `Benutzerdaten`), angetragen an den Ausgang `Berechtigung-prüfen`, abschickt und der andere den gesamten Vorgang abbricht.

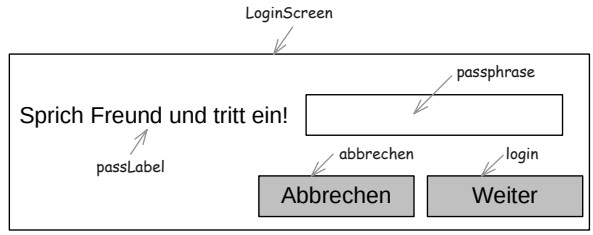


Abb. 9.8 Schematic des Views „LoginScreen“

Abb. 9.8 zeigt zur Verdeutlichung des Aussehens der modellierten Maske ein entsprechendes Schematic (skizzenhafte Darstellung).

Datenbindung

Im Modell aus Abb. 9.7 sind drei Arten von Datenbindung dargestellt, die wir im Folgenden näher untersuchen wollen. Zuerst einmal wird jeder `View` mit dem Modellelement verknüpft, das der zugehörige `ViewCall` als Eingabeparameter erhält (in diesem Falle `Benutzerdaten`). Diese Bindung gilt im Folgenden als das Wurzelelement des Modells, auf dem alle weiteren Bindungen operieren. Die dem `View` untergeordneten Container (z. B. Forms) können eigene Bindungen erhalten, die immer relativ zum übergeordneten Wurzelelement angegeben werden. So ist es möglich die dritte Art Datenbindung, nämlich die der einzelnen Viewelemente zum Datenmodell, immer kontextspezifisch zum entsprechenden *`ContextContainer`* zu spezifizieren, was die Angabe des Bindungspfads erheblich vereinfacht. Die Angabe der Bindung des `ViewElements` mittels des Attributs `binding` geschieht also immer relativ zur Bindung des direkt übergeordneten Containers, dessen seinerseits gebundenes Datenelement als Wurzel des Bindungspfads fungiert.

Im Beispiel ist das Eingabeelement `passphrase` an das gleichnamige Attribut einer Instanz der Klasse `Benutzerdaten` gebunden. Da `Benutzerdaten` das Wurzelelement des übergeordneten Containers `loginForm` darstellt, verkürzt sich die Angabe des Bindungspfads von eigentlich `Benutzerdaten.passphrase` auf `passphrase`. Die Bindung von `Benutzerdaten` an das oberste Containerelement `View` macht deutlich, dass es sich bei `Benutzerdaten` gleichzeitig um den zur Laufzeit als Eingabe erwarteten Input des aufrufenden `ViewCalls` handelt. Abb. 9.9 visualisiert den Sachverhalt grafisch.

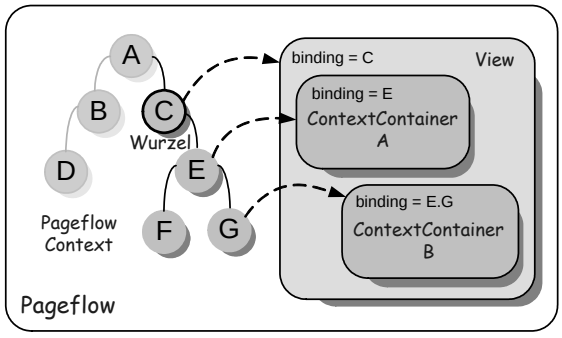


Abb. 9.9 Datenbindung an Containerelemente

Ein View bekommt ein Modellelement des Pageflow-Kontextes übergeben, das für alle weiteren Datenbindungen als Wurzel fungiert (hier Knoten `C`). Die Angabe des Bindungspfads von Container *A* bezieht sich auf diese Wurzel und wird so zu `E`. Auf die gleiche

Weise ergibt sich die Angabe des Pfades der Bindung von Container *B* zu `E.G`.

Bindung von Containern an Ausgänge (Outcomes)

Als letzte Eigenheit des Modells ist noch zu klären, wie festgelegt werden kann, welche Datenwerte aus den Viewelementen (z. B. einem Textfeld) bei auslösen eines bestimmten *Ausgangs* in das unterliegende Modell übernommen und als Nutzdatenpaket ans Ausgangsobjekt angehängt werden sollen. Diese Angabe geschieht durch das Anlegen einer Assoziation, die den entsprechenden Ausgang einem *`ContextContainer`* zuordnet. Wird der Ausgang von einem Benutzer aktiviert, werden die Werte der Datenelemente, die diesem Container zugeordnet sind, mit den Eingaben des Benutzers überschrieben und der `ViewCall` von der Aktivierung benachrichtigt. Der `ViewCall` instanziiert dann das Ausgangsobjekt und hängt das Modellelement, das der betroffenen Form (dem `ContextContainer`) zugeordnet ist, dem Ausgang als Nutzdatenpaket an und liefert es als Rückgabewert des `ViewCalls` zurück.

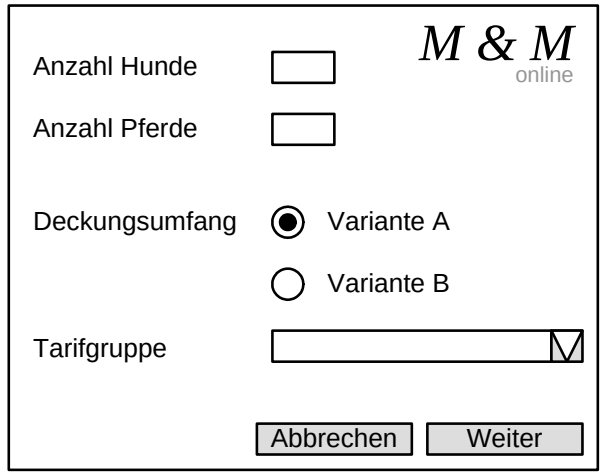
9.2.2 Das erste Modell des Projekts „M&M online“

Werfen wir nun nach der ganzen Theorie einen Blick darauf, wie die eben entwickelten Profile und Metamodelle innerhalb des Application Engineering unseres Projektes praktisch verwendet werden, um die Dialogflüsse des geplanten Systems plattform-unabhängig zu modellieren. Betrachten wollen wir die Aktivität `THH-Tarifierung und Angebotserstellung`. Die Abkürzung „THH“ steht an dieser Stelle für Tierhalterhaftpflicht und realisiert den, in Kapitel 7 identifizierten, GAF `Angebot erstellen` für die Sparte Tierhalterhaftpflicht.

Betrachten wir zuerst die Schematics des Dialogflusses, um ein Gefühl für den darzustellenden Sachverhalt zu bekommen (Abb. 9.10). Angegeben werden kann die Anzahl der Hunde bzw. der Pferde, die der Interessent versichern lassen will sowie der gewünschte Deckungsumfang und die berufliche Tarifgruppe, der er angehört.

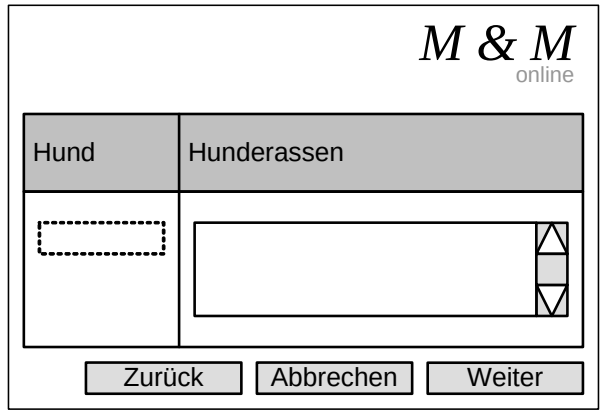
Der weitere Dialogfluss ist abhängig von den hier gemachten Eingaben: sollen ein bzw. mehrere Hunde versichert werden, werden weitere Details zu diesen erfasst (Abb. 9.11). Zu jedem Hund ist in diesem Schritt die Hunderasse aus einer angebotenen Liste auszuwählen.

Abb. 9.10 Schematic zu „Basisdaten erfassen“



Sind alle Angaben zu eventuell zu versichernden Hunden gemacht, kann die Ermittlung der Angebote geschehen. Die Ergebnisse der Anfrage werden in der letzten Seite des Angebotsdialoges dargestellt (Abb. 9.12).

Abb. 9.11 Erfassung der Hunderassen



Zunächst werden die wichtigsten Angaben des Interessenten noch einmal zusammengefasst und dann, am unteren Rand der Maske, die gefundenen Produkte in einer Tabelle dargestellt. Jede Zeile der Tabelle repräsentiert dabei eines der Produkte.

An dieser Stelle hat der Benutzer vier Möglichkeiten zur Interaktion mit dem System: er kann sich für ein Produkt entscheiden und den Antragsprozess starten oder sich zu einem Produkt ein individuelles Angebot erstellen lassen. Weiterhin bestehen die Möglichkeiten den Gesamtvorgang abbzurechen oder im aktuellen Dialogfluss einen Schritt zurückzugehen, um seine Angaben zu verändern.

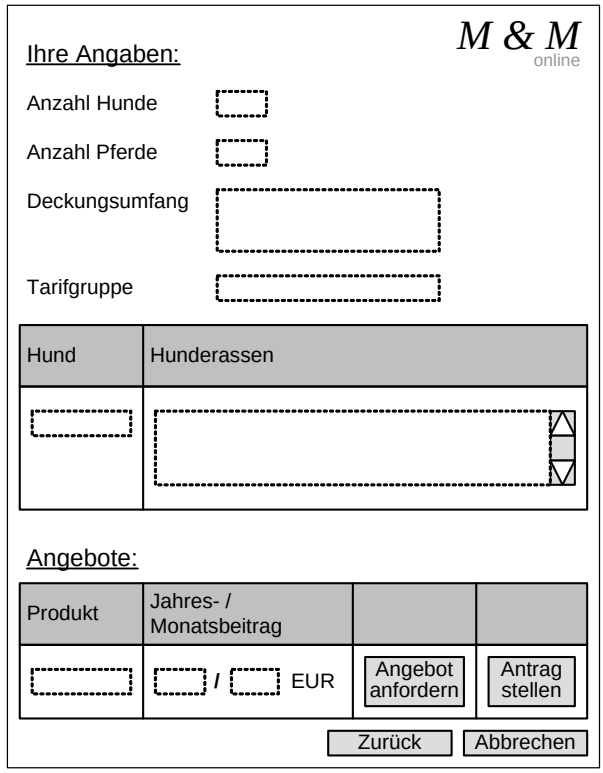


Abb. 9.12 Darstellung der Ergebnisse

Modellierung des Sachverhalts mittels gPIPfP

Soweit zur schematischen Darstellung des Sachverhalts. Wie schauen nun die Ergebnisse der formalen Modellierung mittels gPIPfP aus (Abb. 9.13) ?

Der Dialogfluss beginnt mit dem Aufruf einer Systemaktion, die eine Instanz der Klasse `THHFormular` erstellt und die zugehörigen Felder initialisiert. Die zurückgelieferte Objektinstanz dient dem folgenden `ViewCall` als Eingabe, der seinerseits den in Abb. 9.10 skizzierten Dialog aufruft und diesem das erzeugte Formular als Modell übergibt. Die Modellierung der Viewelemente des Dialogs sowie die Datenbindung der Elemente an das übergebene Modell sind hier nicht dargestellt, sondern finden in einem weiteren Modell statt, das wir in Kürze anschauen werden. Betätigt der Interessent nun den „Weiter“-Button, erreichen wir die erste Verzweigung des Flusses (Anm.: Die Knöpfe „Zurück“ und „Abbrechen“ sind zur Vereinfachung des Diagramms nicht dargestellt. Hier ist grundsätzlich zu überlegen, ob diese „Standardknöpfe“ nicht sowieso rein deklarativ notiert und die Erledigung der notwendigen Aktionen implizit dem Framework überlassen werden sollten.).

Das Ausgangsobjekt `Weiter` führt als Modell das Formular mit sich, das durch Anwendung des `StripModel`-Stereotypen auf die ausgehende Kante des Aufrufs vom Ausgangsobjekt getrennt wird, und so der folgende Entscheidungsknoten direkt auf `THHFormular` arbeiten kann. Sollen ein oder mehr Hunde versichert werden, geht es mit der Erfassung der Hunderassen weiter, ansonsten erfolgt direkt die Ermittlung der relevanten Angebote durch eine Systemaktion namens `Angebote ermitteln`.

Im Dialog, in dem die Ergebnisse dieser Ermittlung dem Interessenten präsentiert werden, hat der Benutzer neben den Standards „Abbrechen“ und „Zurück“ zwei weitere Möglichkeiten der Interaktion: zum einen kann er sich zu einem Produkt ein individuelles Angebot zukommen lassen,was den aktuellen Fluss mit dem Outcome `Angebot anfordern` beendet. Zum anderen kann er sich dazu entschließen, online einen Antrag für ein bestimmtes Produkt zu stellen, was im Outcome `Antrag stellen` resultiert, der das ausgewählte `Angebot` mit sich führt. Dieses `Angebot` sowie das zuvor ausgefüllte `THHFormular` dienen der nun folgenden Systemaktion `Antragsdaten aufbereiten` als Eingabe.

UML ActionSemantics revisited

Geliefert wird `THHFormular` durch eine Aktion vom Typ `ReadStructuralFeatureAction` (≈„Lese ein Attribut“) aus den UML-ActionSemantics (vgl. Kapitel 3). `THHFormular` ist Teil des Pageflow-Kontextes von `THH-Tarifierung und Angeboterstellung` und kann daher von der Aktion gelesen werden. Interessant ist die Art und Weise, wie die Aktion, die dieses Merkmal liest und zurückliefert, aktiviert wird. Wie man sieht, führen keine Kanten in die Aktion hinein, woher „weiß“ die Aktion also, dass sie loslegen kann? Die Antwort auf diese Frage liegt in der speziellen Beschaffenheit des Pins, der `THHFormular` liefert. Es handelt sich um einen `ActionInputPin`, der eine *Pull-Semantik* für Objekt-Token realisiert. Während es im Normalfall so ist, dass Aktionen am Ende ihres Verhaltens Objekt-Token an die entsprechenden Ausgabepins legen und damit die Aktion(en) aktivieren, die weiter hinten im Kontrollstrom liegen (Push-Semantik), aktiviert der `ActionInputPin` seine zugehörige Aktion genau dann, wenn die Ausgabe, die er liefern soll, benötigt wird. Man kann sich das in etwa so vorstellen, dass `Antragsdaten aufbereiten` die Aktion `THHFormular` „auffordert“ ihr Verhalten zu starten, um dann die entsprechenden Ausgaben liefern zu können. Da dies damit zu vergleichen ist, dass die abhängigen Aktionen die benötigten Eingaben aus den entsprechenden Aktionen „herausziehen“, spricht man von einer Pull-Semantik.

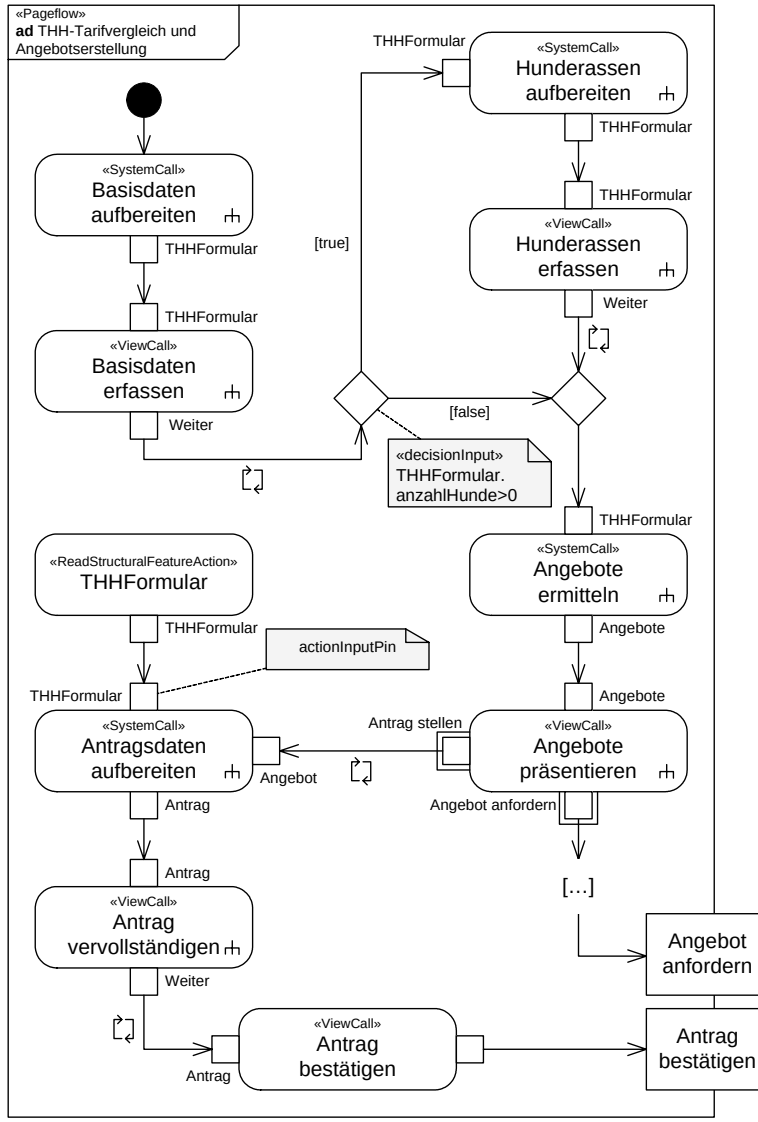


Abb. 9.13 Dialogfluss „Tarifvergleich und Angeboterstellung“

Im weiteren Verlauf des Dialogflusses erhält der Interessent nun noch die Möglichkeit, die Daten, die zur Stellung des Antrags benötigt werden, zu vervollständigen und zu bestätigen. Die Dialogfolge endet mit der Bestätigung des Antrages und dem Ausgang `Antrag bestätigen`, das den ausgefüllten Antrag als `model` mit sich führt und den `Pageflow` beendet.

Detaillierung der Views

Wie bereits besprochen gehören zur Beschreibung eines Dialogflusses noch weitere Details, um die gestellten Anforderungen vollständig abbilden zu können. So wird zumindest noch jeder von einem `ViewCall` aufgerufene `View` weiter detailliert und mit dem unterliegenden Datenmodell verbunden.

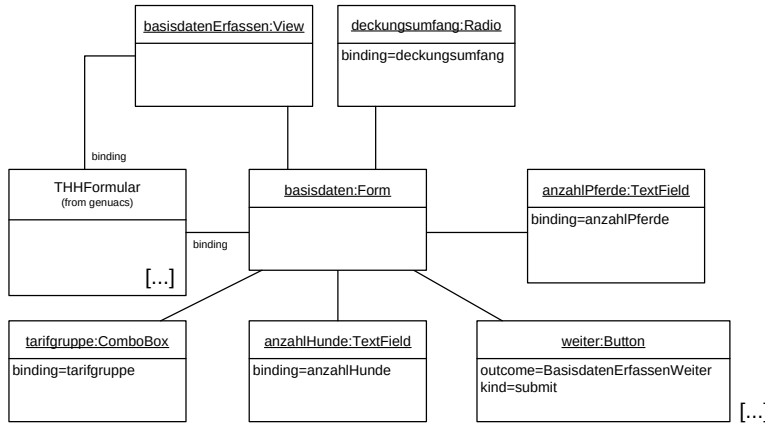


Abb. 9.14 View „Basisdaten erfassen“

Zusätzlich werden die Interaktionsmöglichkeiten, die dem Benutzer innerhalb des Dialogs zur Verfügung stehen, festgelegt. Abb. 9.14 zeigt einen Ausschnitt aus dem Diagramm für den Dialog `Basisdaten erfassen`.

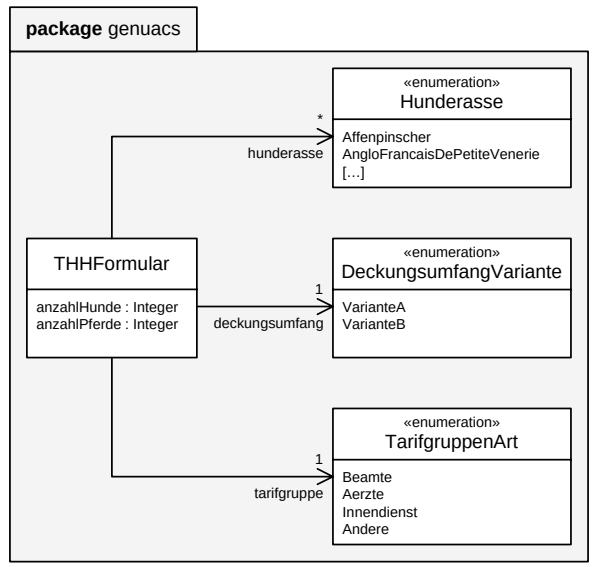


Abb. 9.15 Klassendiagramm des Pageflow-Datenmodells

Die Modellierung der Klasse `THHFormular` geschieht in einem eigenen Klassendiagramm und ist hier nur als Shortcut dargestellt. Das noch fehlende Diagramm liefert (Abb. 9.15).

9.2.3 genua Beehive Pageflow Profile (gBPfP)

Wenden wir uns nun der Bindung unseres gerade aufgestellten gPIPf-Profils auf eine technische Plattform zu. Bereits in *Kapitel 8 – Projektplanung* hatten wir als Zielplattform das Apache Beehive Framework, genauer gesagt die Beehive Pageflows dazu auserkoren. Unsere Wahl haben wir an der angegebenen Stelle mit der umfangreichen Unterstützung von Dialogflüssen durch dieses Web Application Framework begründet. In dem von uns gewählten Transformationsansatz für die Überführung eines in gPIPfP notierten Pageflows in einen Beehive Pageflow werden wir dementsprechend regen Gebrauch von dem angesprochenen Funktionsumfang machen und dabei die Steuerung des Kontroll- bzw. Datenflusses eines Pageflows, also „das Schalten“ der Flüsse (im UML 2 Jargon `ControlFlow` und `ObjectFlow` genannt), vollständig dem Beehive Framework überlassen.

Bevor wir uns der Formulierung des Profils widmen, erinnern wir uns zunächst an die Grundidee von Beehive Pageflows, die darin besteht den Kontroll-/Datenfluss eines Pageflows in hinreichender Weise in einer einzelnen Controller-Klasse zu spezifizieren. Dazu besitzt eine solche Controller-Klasse einer Menge von Java-Methoden, die im Kontext von Beehive *Aktionen* genannt werden. Jede dieser Aktionen gibt ein Objekt vom Typ `Forward` zurück, anhand dessen die nächste aufzurufende Aktion (in diesem Fall spricht man von *Action-Chaining*) oder alternativ eine anzuzeigende Maske (manifestiert als JSP bzw. JSF-View) ermittelt wird.

Die Verbindung zwischen den einzelnen Aktionen und Masken wird also nicht durch einen direkten Aufruf erreicht, sondern erfolgt entkoppelt über eine Art Meta-Objekt. Zur Deklaration eines solchen Meta-Objekt bedient sich Beehive der Annotationen nach JSR-185. Von besonderem Interesse für uns sind in diesem Zusammenhang insbesondere die Annotationen vom Typ `JPF.Action` sowie `JPF.Forward`, die als Bindeglied zwischen Aktionen und Masken bzw. zwischen Aktionen untereinander betrachtet werden können (vgl. Ausführungen in Kapitel Projektplanung).

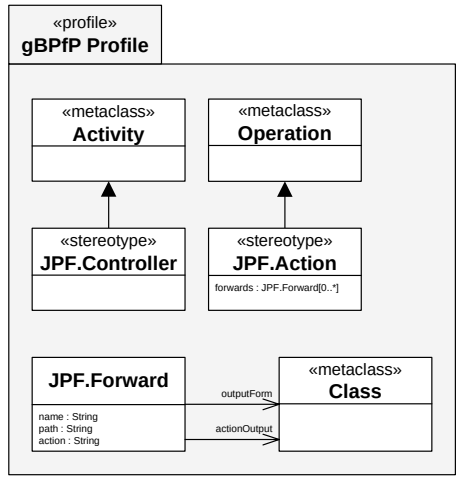


Abb. 9.16 Profil für Dialogfluss mit Beehive Pageflows

Die beschriebene Verkettung der Methoden innerhalb von Beehive erfordert es, dass wir in Bezug auf unser Quellmodell eine zusätzliche Indirektion in unser Zielmodell einführen müssen. Diese Indirektion besteht darin, dass wir für jeden Knoten (im UML 2 Metamodell: `AcitvityNodes`) unseres plattform-unabhängigen Pageflows eine Operation am resultierenden Controller erzeugen müssen. Wie wir noch sehen werden, hat dies allerdings den sehr angenehmen Nebeneffekt, dass dies die Formulierung der Transformationsvorschriften – insbesondere was die Abbildung von Schleifen betrifft – sehr vereinfacht. Abb. 9.16 zeigt den soeben erörterten Zusammenhang in Form eines UML-Profils – das *genua Beehive Pageflow Profile (gBPfP)*. Unser Profil setzt sich zusammen aus:

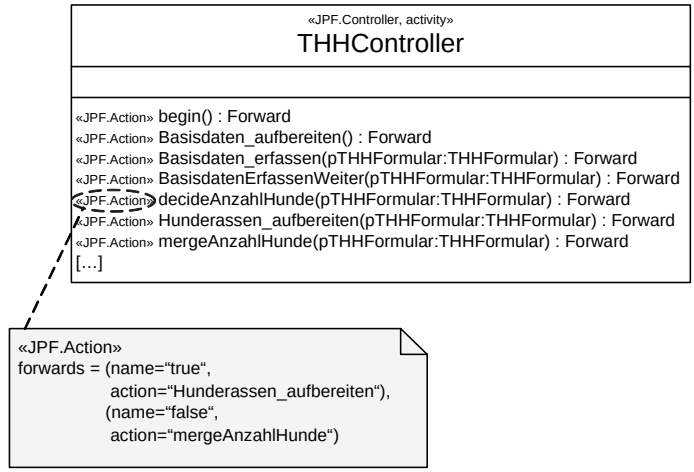
- dem Stereotyp für die bereits erwähnten Controller-Klassen («`JPF.Controller`»), der auf Aktivitäten angewendet wird,
- dem Stereotyp für Aktionen («`JPF.Action`»), der Anwendung auf Operationen findet,
- und der Klasse `JPF.Forward`, deren Instanzen als Rückgabe einer Aktion fungieren. Sie assoziiert andere Klassen über die Attribute `outputForm` und `actionOutput` und verweist auf Folgeaktionen (`actions`) oder Pfaden zu Folgemasken (`paths`).

Mit diesen Basiselementen gerüstet können wir uns nun aufmachen die Tranformationsregeln von gPIPfP- in gBPfP-Modelle zu formulieren:

- Aus einer gPIPfP-Aktivität mit dem Stereotyp «Pageflow» wird eine Ziel-Aktivität mit dem Stereotyp «JPF.Controller» erzeugt.
- Für jede Action einer «Pageflow»-Activity wird eine entsprechende Operation an der Ziel-Aktivität angelegt. Die resultierenden Operationen erhalten den Stereotyp «JPF.Action».
- gPIPfP-Actions mit dem Stereotyp «View Call» resultieren in einem Forward-Objekt mit dem Pfad (path), der auf den im Quellmodell angegebenen View weist (in unserem Fall ist dies immer ein JSF-View mit der Endung .faces). Das Forward-Objekt wird dem forwards-Attribut des Stereotypen «JPF.Action» der betrachteten Ziel-Operation zugewiesen.
- Mit dem Stereotyp «SystemCall» versehene Actions des Quellmodells ergeben hingegen ein Forward-Objekt, bei dem das Attribut action auf eine Folgeaktion referenziert (Action-Chaining). Der ursprünglich spezifizierte Aufruf des Verhaltens (CallBehaviorAction bzw. CallOperationAction) erfolgt aus der erzeugten Operation bzw. Methode. Ein Beispiel dazu werden wir im weiteren Verlauf dieses Kapitels noch sehen.
- DecisionNodes und MergeNodes erhalten ebenfalls ein Pendant in Form einer «JPF.Action»-Operation an der Ziel-Aktivität. Bei einem Entscheidungsknoten wird das spezifizierte Entscheidungsverhalten («decisionInput») aus dieser Operation aufgerufen. Die Operation eines Verbindungsknoten dient hingegen lediglich als Punkt zur Verschmelzung verschiedener Kontrollflüsse.
- Im Fall von «SystemCall»s dienen InputPins direkt als Eingabeparameter des aufzurufenden Systemverhaltens. OutputPins werden in das Forward-Objekt „verpackt“, dass von der entsprechenden JPF-Action zurückgegeben wird. Hierbei kommt eine Fallunterscheidung zum Tragen: im Falle von «SystemCall»s wird das Rückgabeobjekt in das Attribut outputForm des Forward-Objektes geschrieben, bei einem «ViewCall» wird es in analoger Weise in das Attribut actionOutput eingetragen. Unter dem Aspekt der notwendigen Quellcodegenerierung werden wir dieser Fallunterscheidung auf den Grund gehen.

Nach dieser theoretischen der Abbildungsregeln schauen wir uns ihre Anwendung an einem Beispiel an. Beginnen wir zunächst mit der statischen Sicht auf einen resultierenden Pageflow, dargestellt in Abb. 9.17.

Abb. 9.17 Statische Sicht JPF-Controller



Die Aktivität `THHController` ist hier als Klasse notiert (wir entsinnen uns: im UML-Metamodell ist `Activity` eine Spezialisierung von `Class`). Zu den Aktionen aus unserem Quellmodell wie etwa „`Basisdaten aufbereiten`“, „`Basisdaten erfassen`“, usw. (vgl. Abb. 9.13) sind jeweils Operationen am Ziel-Controller generiert worden (Anmerkung: Leerzeichen “ “ im Namen wurden hier durch einen Unterstrich “_“ ersetzt). Darüber hinaus sind Operationen für den Beginn eines Pageflows (`begin`), den Wiedereinstieg aus dem Outcome eines Views (z. B. `BasisdatenErfassenWeiter`) sowie für Entscheidungs- bzw. Verbindungsknoten (exemplarisch hier: `decideAnzahlHunde` respektive `mergeAnzahlHunde`) erstellt worden.

Allen Operationen gemein ist, dass sie mit dem Stereotyp `«JPF.Action»` versehen sind, welcher wiederum die Folgeaktionen/-masken über das Attribut `forwards` spezifiziert. In Abb. 9.17 ist dies exemplarisch für die Methode `decideAnzahlHunde` dargestellt: hier sind Verweise auf die Folgeaktionen `mergeAnzahlHunde` und `Hunderassen_erfassen` festgehalten.

Wie bereits weiter oben angedeutet, implizieren die in den `forwards` angegebenen Informationen weitgehend das Verhalten eines Pageflows, da sie die verschiedenen Aktionen bzw. Masken des selbigen miteinander verbinden. Die in Abb. 9.17 dargestellte statische

Sicht beschreibt einen Beehive Pageflow also schon zu einem wesentlichen Teil. Die Verhaltenssicht eines Controllers kann aber zusätzlich expliziert werden, indem man Kontroll- und Datenflüsse analog zu unserem plattform-unabhängigen Modell sichtbar macht.

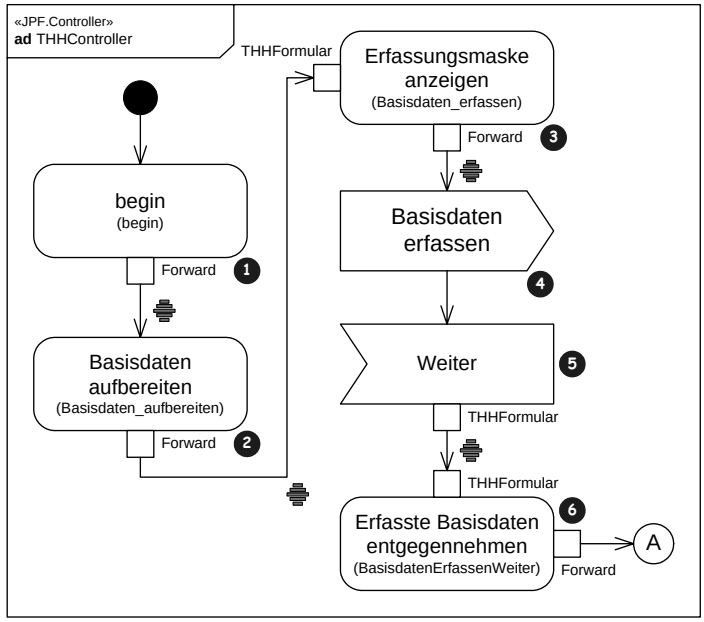


Abb. 9.18 Verhaltenssicht JPF-Controller Teil 1

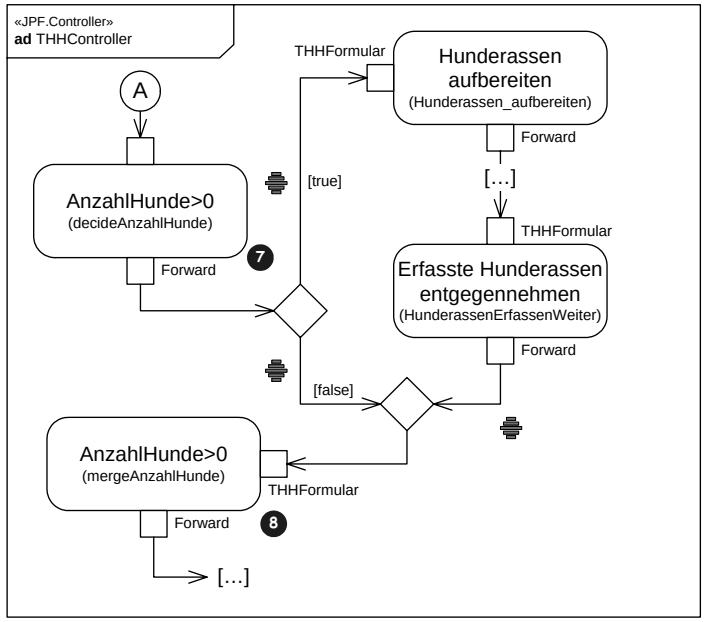
Abb. 9.18 zeigt dazu wiederum einen Ausschnitt aus dem erzeugten JPF-Controller. Bei den angegebenen Aktionen ❶,❷,❸ und ❻ handelt es sich um jeweils `CallOperationActions`, also um Operationsaufrufe. Diese rufen die in Abb. 9.17 gelisteten `«JPF-Actions»` auf der Name der entsprechenden Operation ist dabei jeweils in Klammern innerhalb des Aktionssymbols notiert.

Der Aufruf eines Views innerhalb von Beehive erfolgt nicht im eigentlichen Sinne synchron, sondern bedingt durch den HTTP-Request/Response-Zyklus asynchron. Aufruf und Rückkehr haben wir daher durch eine `SendSignalAction` (❹) und eine `AcceptEventAction` (❺) symbolisiert. Durch Betätigung des Command-Buttons mit der `id="weiter"` auf der Erfassungsmaske wird ein Ereignis gefeuert, welches von der `AcceptEventAction` entgegengenommen wird. Im Anschluss daran wird der Kontrollfluss durch den Aufruf der Operation `BasisdatenErfassenWeiter` (❻) fortgesetzt.

Besonderer Erwähnung bedürfen die Kanten, die mit einem Bienenstock-Symbol (🐝) behaftet (stereotypisiert) sind. An ihnen kommt die folgende „Beehive-Magie“ ins Spiel: Zur Laufzeit entpackt der Beehive Controller das eventuell vorhandene Nutzdatenpaket (`outputForm` bzw. `actionOutput`) und ermittelt über *Reflection* die aufzurufende Folge-Methode mit der passenden Signatur.

Vertiefen wir unseren Diskurs noch etwas und schauen uns Abb. 9.19 an. Die Abbildung widmet sich insbesondere der Veranschaulichung der Übersetzung von `DecisionNodes` und `MergeNodes` des plattform-unabhängigen Quellmodells (vgl. Abb. 9.13) in explizite Operationen des plattform-spezifischen Zielmodells.

Abb. 9.19 Verhaltenssicht JPF-Controller Teil 2



In unserem Beispiel heißen die beiden resultierenden Operationen `decideAnzahlHunde` (❼) und `mergeAnzahlHunde` (❽). Die Folgeaktion der *„Entscheidungsaktion“* wird anhand des zurückgelieferten Forward-Objekts durch den Beehive-Controller bestimmt. Das Forward-Objekt trägt dazu als Namen das Resultat des Enscheidungsverhaltens – in unserem Fall also `"true"` oder `"false"`.

Vom PSM zum Code

Bleibt schließlich die Überführung in lauffähigen Quellcode. Abb. 9.20 zeigt einen Ausschnitt aus einem entsprechenden Code-Artefakt.

```
// Ausschnitt aus JPF-Controller
@Jpf.Controller                                              (1)
public class THHController extends PageFlowController {

  @Control private GenuaServices mGS;                        (2)

  @Jpf.Action(forwards = {                                   (3)
    @Jpf.Forward(name="default",                             (4)
                 action="Basisdaten_aufbereiten")})
  public Forward begin() {
    return new Forward("default");
  }

  @Jpf.Action(forwards = {
    @Jpf.Forward(name="default",
                 action="Basisdaten_erfassen")})
  public Forward Basisdaten_aufbereiten() {                  (5)
    THHFormular lTHHFormular =
      mGS.basisdatenAufbereiten();                           (6)
    Forward lForward = new Forward("default");
    lForward.addOutputForm(lTHHFormular);                    (7)
    return lForward;                                         (8)
  }

  @Jpf.Action(forwards = {
    @Jpf.Forward(name="default",
                 path="Basisdaten_erfassen.faces")})         (9)
  public Forward Basisdaten_erfassen(
    THHFormular pTHHFormular) {
    Forward lForward = new Forward("default");
    lForward.addActionOutput("THHFormular",
                             pTHHFormular);                 (10)
    return lForward;
  }

  @Jpf.Action(forwards = {
    @Jpf.Forward(name="default",
                 action="decideAnzahlHunde")})
  public Forward BasisdatenErfassenWeiter(                   (11)
    THHFormular pTHHFormular) {
    // Create Outcome                                        (12)
    BasisdatenErfassenWeiter lBasisdatenErfassenWeiter =
      new BasisdatenErfassenWeiter(pTHHFormular);
```

Abb. 9.20 Ausschnitt Quelltext JPF-Controller

Abb. 9.20 Ausschnitt Quelltext JPF-Controller (Forts.)

```
    // Transformation / Strip Model                    (13)
    THHFormular lTHHFormular =
      (THHFormular)lBasisdatenErfassenWeiter.getModel();
    Forward lForward = new Forward("default");
    lForward.addOutputForm(lTHHFormular);
    return lForward;
  }
  […]
}
```

Die Annotation `JPF.Controller` (1) weist die dargestellte Klasse `THHController` als Controller-Klasse aus und entspricht somit dem gleichnamigen Stereotypen (vgl. Abb. 9.16). Dies gilt gleichermaßen für die unter (3) dargestellte `JPF.Action`. Entsprechende Foward-Deklarationen (in unserem UML-Profil eigens als `Foward`-Klasse modelliert) werden gleichermaßen als Annotationen umgesetzt (4). Die unter dem Punkt angegebene `«JPF.Action»`-Methode `begin` ruft beispielsweise die Folgeaktion `Basisdaten_aufbereiten` auf.

In dieser Methode (5) wird zunächst ein Objekt vom Typ `THHFormular` erzeugt (6). Dies geschieht über den Aufruf einer Operation eines zuvor deklarierten (2) Beehive Controls. Die Interna dieser Service-Schnittstelle sowie ihre Instanziierung interessieren uns an dieser Stelle nicht, stattdessen verweisen wir zu weiteren Details auf den noch folgenden Abschnitt 9.3. Im Anschluss an diesen Aufruf wird die Instanz als Attribut `outputForm` in das `Forward`-Objekt geschrieben (7), dass von der Methode zurückgegeben wird (8).

Die unter (9) dargestellte Aktion bringt eine Maske zur Anzeige (`path`-Attribut zeigt auf einen JSF-View). Im Gegensatz zur zuvor beschriebenen `«JPF.Action»`-Methode wird das Formular hier als `actionOutput` dem Rückgabe-Objekt zugewiesen (10).

Bleibt schließlich die Methode `BasisdatenErfassenWeiter` (11). Wie wir bereits weiter oben diskutiert hatten, wird sie durch das Betätigen des `"weiter"`-Buttons ausgelöst. In der anschließenden Diskussion des plattform-spezifischen View-Modells werden wir noch sehen, wie sich die zugehörige Verknüpfung gestaltet.

An der dargestellten Methode ist darüber hinaus ersichtlich, wie das Outcome- und das Strip Model-Konzept umgesetzt werden. Unter (12) wird ein Outcome-Objekt erzeugt, das Nutzdatenpaket (unser `THHFormular`) wird dabei dem Konstruktor als Eingabeparameter übergeben. Das Outcome-Objekt wird an der ausgehenden

Kante allerdings nicht weiter benötigt, das Nutzdatenobjekt wird daher direkt wieder heraus gestripped (13). (Anm.: In einer späteren Version der gPIPfP-zu-gBPfP-Transformation sollten solche redundanten Schritte optimiert werden).

9.2.4 genua JSF Metamodel (gJSFMM)

Analog zu Abb. 9.14 zeigt Abb. 9.21 die plattform-spezifische Sicht auf die Maske zur Erfassung der Basisdaten aus unserem Fallbeispiel. Da wir uns initial auf ein Minimum von grafischen Elementen beschränkt haben, ergeben sich die Transformationsregeln gerade heraus, d. h. die Elemente aus unserem plattform-unabhängigen View Metamodell werden „eins zu eins“ auf entsprechende Elemente des JSF-Frameworks abgebildet.

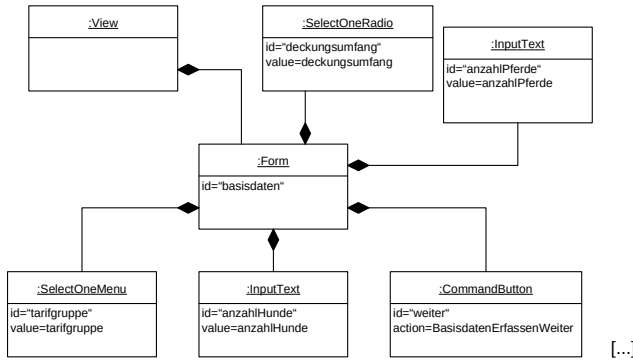


Abb. 9.21 JSF-Komponentenbaum

Die Umformung eines JSF-PSMs in die textuelle Notation beschränkt sich auf das Layout des Quellcodes. Betrachten wir Abb. 9.22, finden wir dort unseren View (2) samt des eingebetteten Formulars (3) und der darunter angeordneten visuellen Komponenten, beispielsweise das Eingabefeld zur Erfassung der Anzahl der zu versichernden Hunde (4).

Besonderer Erläuterung bedarf der Weiter-Button unseres Views (6). Er liefert kein Ergebnis-Objekt zurück, wie dies unser plattformunabhängiges Modell vermuten ließe, sondern ruft eine gleichnamige Methode am JPF-Controller auf (7) (vgl. Ausführungen zu Beehive Pageflows in Kapitel 8). Die Nutzdaten, die der entsprechenden Methode als Eingabeparameter dienen, werden über das Attribut `submitFormBean` (8) transportiert.

Abb. 9.22 Ausschnitt Quelltext JSF-View

```
[…]
<html>
<head>
  <title>genuacs - Basisdaten erfassen</title>
  <link rel="stylesheet" type="text/css"
        href="Basisdaten_erfassen.css"/>                    (1)
</head>
<%@taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<f:view>                                                    (2)
  <h:form id="basisdaten">                                  (3)
    […]
    <h:outputText id="anzahlHundeLabel"
                  value="Anzahl Hunde"
                  styleClass="anzahlHundeLabel"/>
    <h:inputText id="anzahlHunde"                           (4)
                 value="#{backing.THHFormular.anzahlHunde}"
                 styleClass="anzahlHunde"/>                 (5)
    <h:outputText id="anzahlPferdeLabel"
                  value="Anzahl Pferde"
                  styleClass="anzahlPferdeLabel"/>
    <h:inputText id="anzahlPferde"
                 value="#{backing.THHFormular.anzahlPferde}"
                 styleClass="anzahlPferde"/>
    […]
    <h:commandButton id="weiter"                            (6)
                     value="Weiter"
                     action="BasisdatenErfassenWeiter"      (7)
                     styleClass="weiter">
      <f:attribute name="submitFormBean"
                   value="backing.THHFormular" />           (8)
    </h:commandButton>
  </h:form>
</f:view>
[…]
```

Der dargestellte Code legt die hierarchische Anordnung der View-Elemente fest, Informationen zu ihrem konkreten Layout sind hier nicht zu finden. Sie werden stattdessen in einer eigenen CSS-Datei festegelgt, die aus der JSP-Datei referenziert wird (1). Eine solche CSS-Datei ist in Abb. 9.23 dargestellt.

```
[…]
.anzahlHundeLabel {left: 8px;                          (1)
                   top: 80px;                          (2)
                   position: absolute;}
.anzahlHunde {left: 120px;
              top: 80px;
              position: absolute;}
.anzahlPferdeLabel {left: 8px;
                    top: 112px;
                    position: absolute;}
.anzahlPferde {left: 120px;
               top: 112px;
               position: absolute;}
[…]
```

Abb. 9.23 CSS-Informationen

Der vorgestellte Ansatz hat den Vorteil, dass der Inhalt der Maske weitestgehend vom Layout getrennt wird. Dies macht es z. B. möglich, dass die Layout-Arbeiten von einer spezialisierten Medienagentur übernommen werden können.

```
[…]
<form id="basisdaten"
      method="post"
      action="/thhtarifvergleich/Basisdaten_erfassen.faces">
  <img id="basisdaten:logo"
       src="/images/m_u_m.png" class="logo" alt="" />
  <span id="basisdaten:anzahlHundeLabel"
        class="anzahlHundeLabel">Anzahl Hunde</span>
  <input id="basisdaten:anzahlHunde"
         type="text"
         name="basisdaten:anzahlHunde"
         value="1"
         class="anzahlHunde" />
  […]
</form>
[…]
```

Abb. 9.24 Aus JSF generiertes HTML

Die Festlegung der X/Y-Koordination (in der Abbildung `left` (1) und `top` (2)) erfolgt über so genannte Style-Klassen. Naheliegend wäre die Verwendung der Id über den #-Selektor gewesen. JSF verhindert dies allerdings, denn die Generierung von (Client-)Ids in JSF ist inkompatibel mit dem CSS-Standards. So verwendet JSF einen Doppelpunkt zur Abgrenzung von Naming-Containern in Ids (siehe

Abb. 9.24). Der angesprochene Workaround besteht nun darin, (style)Classes zu nutzen (mit einem Punkt („.“) als Selektor).

9.2.5 genua Platform Independent Workflow Profile (gPIWfP)

Nachdem wir in den letzten Abschnitten Lösungen für die Modellierung und technische Realisierung von Dialogflüssen gefunden haben, wollen wir uns nun um den nächsten Punkt auf der Liste kümmern: der Darstellung und technischen Abbildung von Workflows.

Innerhalb dieser Betrachtungen wollen wir Workflows als den „Rahmen“ betrachten, der die zuvor einzeln definierten Dialogflüsse zu einer die Geschäftsprozesse unterstützenden Folge von Aktionen aneinanderreiht. Wird ein Prozess gestartet, tauchen in der Arbeitsliste der zuständigen Bearbeiter die zu erledigenden Aufgaben in Form aktivierbarer Links auf. Bei Aktivierung des Links wird der unterliegende Dialogfluss gestartet, der die Erledigung der Aufgabe automatisiert unterstützt. Je nach Ergebnis des Arbeitsschrittes (bspw. Aufgabe erfolgreich erledigt/abgebrochen/…) wird der übergeordnete, kontextgebende Workflow weitergeführt, was unter Umständen in der Erzeugung neuer Aufgaben resultiert, die dann wiederum den zuständigen Bearbeitern zugeordnet werden.

Während Workflows zwischen ihren Schritten persistenten Charakter besitzen, werden Dialogflüsse im Allgemeinen atomar behandelt. Wird eine Aufgabe in einer Sitzung nicht vollständig erledigt gehen (ohne zusätzliche Features der Anwendung wie Laden/Speichern etc.) alle bereits innerhalb des nicht erledigten Arbeitsschrittes erfassten Informationen verloren. Für weitere Aspekte dieses Gedankens möchten wir an dieser Stelle auf Kapitel 8 verweisen, da wir an dieser Stelle bereits ausführlich auf die Unterschiede der verschiedenen atomaren Ebenen eingegangen sind.

Näher betrachtet handelt es sich bei den einzelnen Steps um Aufrufe von `«Pageflow»`-Aktivitäten, also um Aufrufe von Dialogflüssen, die den einzelnen Workflowschritten über ihr `pageflow` Attribut zugeordnet werden.

Wie bei den Dialogflüssen wird das Ergebnis der einzelnen Schritte dem Workflow-Controller über einen Ausgang mitgeteilt (vgl. Abschnitt 9.2.1), das im genuacs-Profil enthalten ist, und von dort importiert wird.

Abb. 9.25 zeigt das plattform-unabhängige Profil, das innerhalb *M&M-Online* erarbeitet und im Folgenden zur Modellierung verwendet wird. Auch hier dient als Basis das UML2-Aktivitätsdia-

gramm. Zur Realisierung der gestellten Anforderungen müssen lediglich zwei Elemente erweitert werden. Zusätzlich kommen „alte Bekannte“ in Form von Modellelementen und Konzepten zum Tragen, die wir bereits aus unseren Betrachtungen zu Dialogflüssen in den vorangegangenen Abschnitten kennen gelernt haben.

Basiskonzepte

Workflows bestehen aus einer Abfolge von *Aktionen* in Form von «WorkflowStep»s, die aneinander gereiht werden, um so zusammen einen Geschäftsprozess abzubilden. Die Steuerung des Workflow-Kontrollflusses übernimmt die übergeordnete Aktivität, die mit dem Stereotyp «Workflow» versehen wird. Zur Modellierung bedingter Kontrollflüsse stehen uns alle Mittel der UML-Aktivitätsdiagramme zur Verfügung (Entscheidungsknoten, Parallelisierungen, …), wodurch keine zusätzlichen Erweiterungen durch das Profil in diese Richtung notwendig werden.

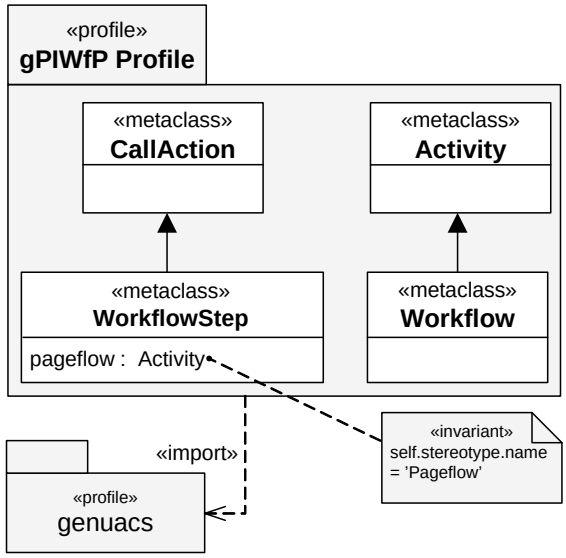


Abb. 9.25
gPIWfP Profil

Näher betrachtet handelt es sich bei den einzelnen Steps um Aufrufe von «Pageflow»-Aktivitäten, also um Aufrufe von Dialogflüssen, die den einzelnen Workflowschritten über ihr `pageflow` Attribut zugeordnet werden.

Wie bei den Dialogflüssen wird das Ergebnis der einzelnen Schritte dem Workflow-Controller über einen Ausgang mitgeteilt (vgl. Abschnitt 9.2.1), das im genuacs-Profil enthalten ist, und von dort importiert wird.

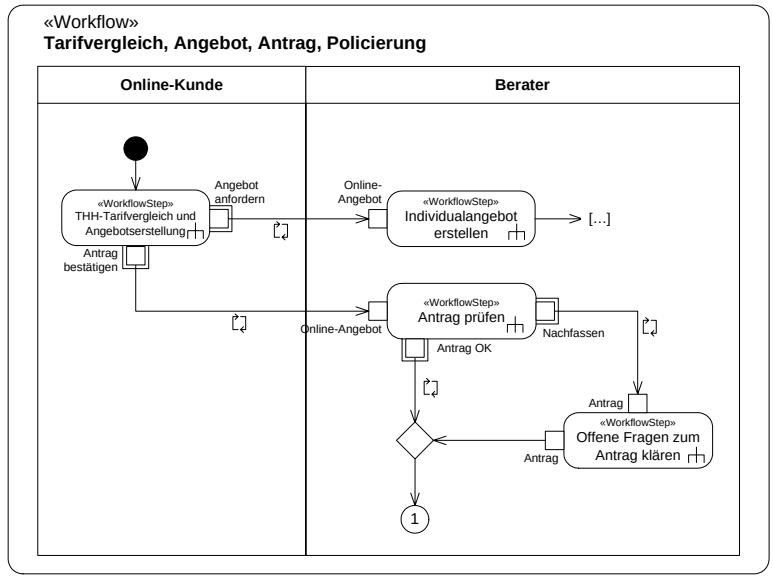
9.2.6 Ein weiteres Modell des M&M-Projektes

Abb. 9.26 zeigt den Workflow, der die Vorgänge Tarifvergleich, Angebot, Antrag und Policierung im Bereich Tierhalter-Haftpflicht (THH) in einen gemeinsamen Kontext setzt und die Anforderung nach einer Online-Realisierung des Geschäftsprozesses umsetzt. Betrachtet werden die beteiligten Rollen Online-Kunde und Berater, deren Aufgaben in der jeweiligen Swim-Lane modelliert wurden.

Der Workflow beginnt mit der Ermittlung eines geeigneten Tarifs durch den Kunden selbst, der durch den Pageflow `THH-Tarifierung und Angebotserstellung` unterstützt wird (vgl. Abschnitt 9.2.2). Je nach Ergebnis des Dialogflusses wird dann vom Berater ein Individualangebot erstellt (in diesem Modell nicht näher detailliert) oder der online vom Kunden gestellte Antrag noch einmal manuell vom Berater auf Gültigkeit geprüft.

Sind noch offene Fragen zu klären, fasst der Berater (telefonisch, per Mail/Post, …) beim Kunden nach, um diese zu klären. Jeder dieser drei zu erledigenden Aufgaben erzeugt jeweils einen Eintrag in der Arbeitsliste des Beraters. Werden die Aufgaben durch einen Dialogfluss unterstützt, wird bei Anwahl einer Aufgabe durch den Berater der entsprechende Dialogfluss gestartet.

Abb. 9.26 Workflow „Tarifvergleich, Angebot, Antrag & Policierung“



Bei nicht durch das System unterstützten Aufgaben, verbirgt sich hinter dem Link lediglich ein Dialog, in dem der erfolgreiche bzw.

nicht erfolgreiche Abschluss der Aufgabe vermerkt werden kann, um den Workflow fortzuführen.

Wie in den Dialogflüssen sind auch in diesem Modell Standard-Ergebnisse wie das komplette Abbrechen des Vorgangs (zum Beispiel durch den Kunden vor der Antragstellung) nicht explizit modelliert, und werden vom System realisiert und angeboten.

Abb. 9.27 führt den Workflow mit den weiteren Aufgaben des Bearbeiters fort. Sind alle offenen Fragen geklärt, kann er für den Kunden beim jeweiligen Versicherer den Versicherungsschutz beantragen. Erhält er nach Ablauf einer zuvor gesetzten Frist keine Antwort (modelliert durch eine Time-Out-Exception), erfolgt eine Erinnerung zum Nachfassen. Bei Erhalt der Antwort bestehen zwei Möglichkeiten: der Versicherer bestätigt den Antrag und schickt eine entsprechende Police oder er lehnt die Versicherung des Kunden ab.

Im Negativfall wird der Kunde von der Entscheidung des Versicherers benachrichtigt und der Workflow wird beendet. Ansonsten leitet der Bearbeiter die Police an den Kunden weiter.

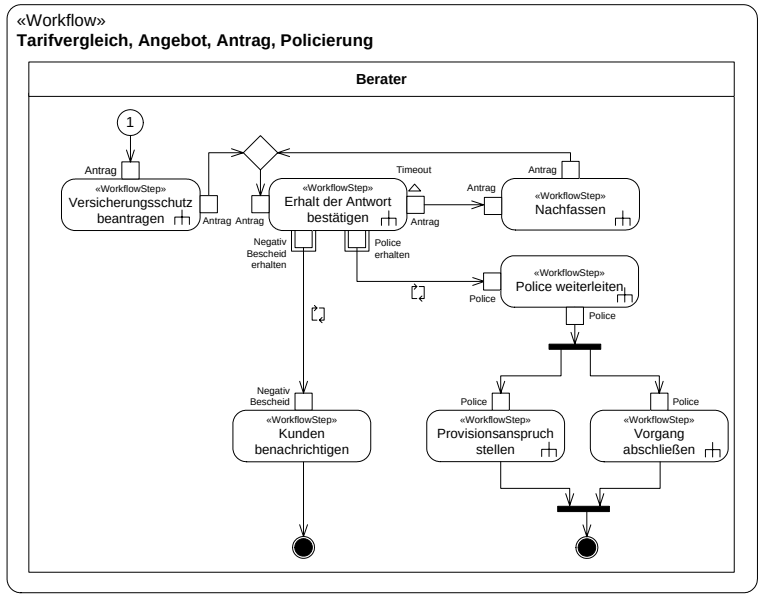


Abb. 9.27 Workflow „Tarifvergleich, Angebot, Antrag & Policierung“ (Forts.)

Nach erfolgter Weiterleitung kann nun der Anspruch auf Provision beim Versicherer geltend gemacht und der Gesamtvorgang abgeschlossen werden.

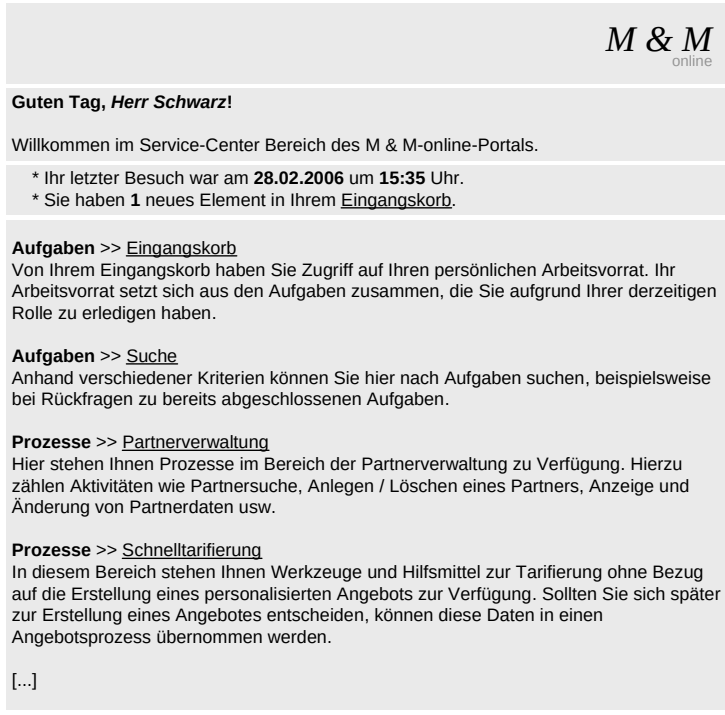
9.2.7 genua jBPM Profil

Schauen wir uns als nächstes an, wie der im vorigen Abschnitt modellierte Workflow auf plattform-spezifischer Ebene in genua umgesetzt wurde. Dazu werden zuerst die Basiskonzepte der Workflowsteuerung innerhalb des Frameworks erläutert und danach das zugehörige UML-Profil vorgestellt. Am Schluss dieses Abschnittes wollen wir dann noch einige Details in Form von Auszügen aus den generierten Artefakten vorstellen, um auch die tatsächliche Implementierung besser zu verstehen. Dabei sollen die im Kapitel über die Projektplanung (Kapitel 8) erläuterten Mechanismen von jBPM nicht wiederholt sondern nur auf Feinheiten eingegangen werden, die wir so noch nicht kennen gelernt haben.

Basiskonzept

Workflows werden in genua durch die Verwendung so genannter „Arbeitslisten“ realisiert, die die Sachbearbeiter auf ihrer persönlichen Arbeitsfläche dem „Desktop“ verwalten können.

Abb. 9.28 Einstiegsseite des M&M-online-Portals


M & M
online
Guten Tag, Herr Schwarz!
Willkommen im Service-Center Bereich des M & M-online-Portals.
* Ihr letzter Besuch war am 28.02.2006 um 15:35 Uhr.
* Sie haben 1 neues Element in Ihrem Eingangskorb.
Aufgaben >> Eingangskorb
Von Ihrem Eingangskorb haben Sie Zugriff auf Ihren persönlichen Arbeitsvorrat. Ihr Arbeitsvorrat setzt sich aus den Aufgaben zusammen, die Sie aufgrund Ihrer derzeitigen Rolle zu erledigen haben.
Aufgaben >> Suche
Anhand verschiedener Kriterien können Sie hier nach Aufgaben suchen, beispielsweise bei Rückfragen zu bereits abgeschlossenen Aufgaben.
Prozesse >> Partnerverwaltung
Hier stehen Ihnen Prozesse im Bereich der Partnerverwaltung zu Verfügung. Hierzu zählen Aktivitäten wie Partnersuche, Anlegen / Löschen eines Partners, Anzeige und Änderung von Partnerdaten usw.
Prozesse >> Schnelltarifierung
In diesem Bereich stehen Ihnen Werkzeuge und Hilfsmittel zur Tarifierung ohne Bezug auf die Erstellung eines personalisierten Angebots zur Verfügung. Sollten Sie sich später zur Erstellung eines Angebotes entscheiden, können diese Daten in einen Angebotsprozess übernommen werden.
[...]

Dort sind zum einen offene Aufgaben, die dem Bearbeiter zugewiesen wurden, aufgeführt und können von diesem zur Weiterbearbei-

tung geöffnet werden. Zum anderen besteht auch die Möglichkeit zur Verwaltung von Workflows, worunter zum Beispiel das Starten weiterer Kontrollflüsse, das Ablehnen bzw. Delegieren von Aufgaben usw. gehören.

Bevor der Benutzer seinen Desktop öffnen kann, muss er sich am System authentifizieren. Dazu gibt er, in Form eines Benutzernamens und Passworts, seine Authentifizierungsangaben in der Anmeldemaske ein. Ist der Anmeldeversuch erfolgreich, startet das System die Arbeitsfläche des Benutzers.

Abb. 9.28 zeigt ein Beispiel dafür, wie ein solcher Desktop im Falle der M&M-online Applikation aussehen kann. Neben den Informationen über den letzten Besuch, wird dem Benutzer die Anzahl der Aufgaben angezeigt, die ihm seitdem zugewiesen worden sind. Der darunter ausschnittweise dargestellte Bereich gliedert sich in zwei Bereiche: zum einen können die aktuell zugewiesenen Aufgaben aufgelistet werden – man spricht in diesem Zusammenhang von der Darstellung des Eingangskorbs des Mitarbeiters – zum anderen können neue Aufgaben erzeugt bzw. neue Prozesse angestoßen werden.

M & M online

Von Ihrem Eingangskorb haben Sie Zugriff auf Ihren persönlichen Arbeitsvorrat. Ihr Arbeitsvorrat setzt sich aus den Aufgaben zusammen, die Sie aufgrund Ihrer derzeitigen Rolle zu erledigen haben.

Status	Kurzbeschreibung	Erstellt am Erstellt von	Spätester Start Spätestes Ende
neu	Antrag prüfen, THH, E. Meier	01.03.2006 online Antrag	-- 06.03.2006
über-fällig	Erhalt der Antwort bestätigen, Antrag auf THH, Consulta, S. Müller	01.02.2006 K. Petersen	-- 01.03.2006

[...]

Abb. 9.29 Ausschnitt einer Aufgabenliste

Dieses Anstoßen von Prozesses resultiert dann wieder in der Erzeugung neuer Workflows und der damit verbundenen zu erledigenden Aufgaben, die dann vom System automatisch den verfügbaren Bearbeitern zugewiesen werden und in Einträgen in den entsprechenden Arbeitslisten resultieren.

Betrachten wir zuerst den Fall, dass der Bearbeiter eine Übersicht über die Einträge seiner Arbeitsliste aufrufen möchte. Ein Klick auf den entsprechenden Link liefert die folgende Ansicht (Abb. 9.29).

Als ersten Eintrag sehen wir den neuen Eintrag in die Liste, der eine Aufgabe des Typs „Antrag prüfen“ enthält und dem gleichnamigen Workflow-Schritt aus dem Workflow in Abb. 9.26 entspricht. Weiterhin zu erkennen ist, dass der Antrag online eingegangen ist, da die Aufgabe vom Online-System erzeugt wurde. Darunter dargestellt eine als „überfällig“ gekennzeichnete Aufgabe, die unserem Bearbeiter von einem Kollegen zugewiesen worden ist.

Beide Aufgaben können zur weiteren Bearbeitung angeklickt werden. Geschieht dies, wird der darunterliegende Dialogfluss gestartet, der mit den bereits aus den vorigen Abschnitten bekannten Konzepten und Technologien modelliert und realisiert wird. Daher wollen wir an dieser Stelle nicht tiefer auf die Aufgaben des Beispiels eingehen, sondern vielmehr betrachten, welche Mittel uns das genua-Framework zur Verfügung stellt, um Workflows mittels jBPM zu realisieren, und wie die Darstellung dieser Realisierung in den entsprechenden plattform-spezifischen Modellen aussieht.

UML-Profil

Abb. 9.30 zeigt einen Ausschnitt des Profils, mit dessen Hilfe die jBPM-spezifische Realisierung der zuvor plattform-unabhängig modellierten Workflows nach der Transformation dargestellt werden soll. Hier wurde vom DE-Team des genua-Frameworks das Zustandsdiagramm zur Darstellung gewählt.

Workflows (genauer jBPM-Workflows) werden als Zustandsautomaten modelliert, in denen Aufgabenknoten (`TaskNodes`) die Erzeugung einer Aufgabe in der Aufgabenliste eines Bearbeiters symbolisieren. Erreicht ein Workflow also einen Aufgabenknoten, wird eine Instanz der Aufgabe erzeugt und der Aufgabenliste des zuständigen Bearbeiters angehängt. Die Ermittlung des Bearbeiters übernimmt der `assignmentHandler` eines Knotens. Die Spezifizierung dieses Handlers geschieht durch die Angabe einer Java-Klasse, die diese Aufgabe übernehmen soll. Die Aufgabe, die es zu lösen gilt, besteht nun darin einen verfügbaren Bearbeiter zu finden, der die in `role` deklarierte Rolle besitzt.

jBPM-Workflows besitzen die Möglichkeit, Prozessvariablen zu verwalten, die während der Durchführung des Gesamtprozesses persistiert werden und so allen Aufgaben zur Verfügung stehen. Die Variablen können über den `context` des Workflows bezogen wer-

den. Der Kontrollfluss des Workflows ist in der jBPM eigenen Process Definition Language (PDL) abgelegt und besteht aus (mindestens) einem XML-File.

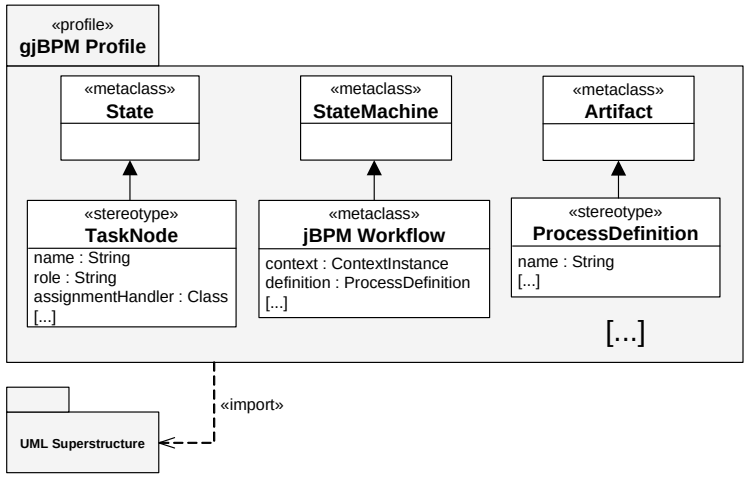


Abb. 9.30 Ausschnitt des gjBPM-Profils

Schauen wir uns nun an, wie ein Ausschnitt aus dem plattformunabhängigen Modell aus Abb. 9.26 in die jBPM-Variante transformiert wurde, um die Zusammenhänge besser zu verstehen.

9.2.8 Transformation von gPIWfP nach gjBPM

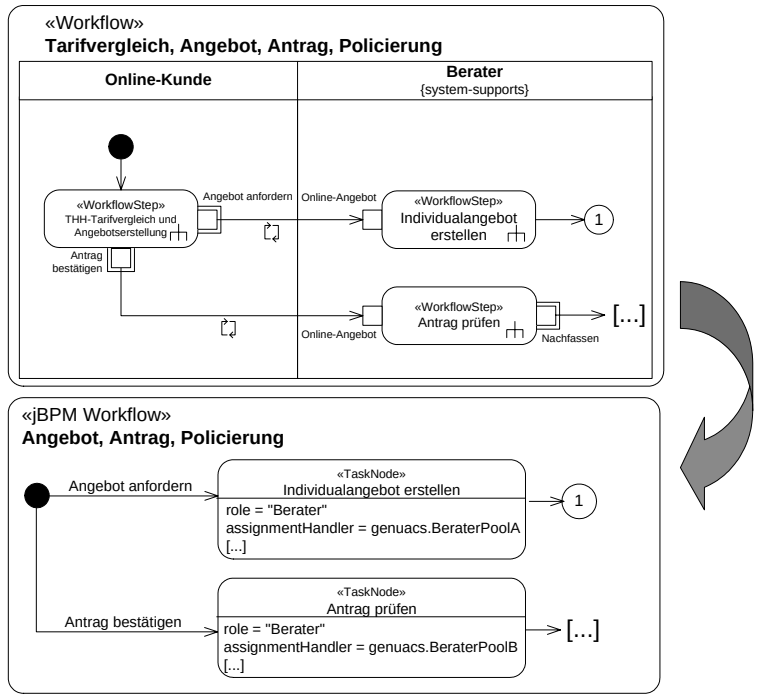
Um einen Eindruck davon zu bekommen, welche Vorgehensweise hinter der Transformation steckt, betrachten wir die relevanten Ausschnitte aus beiden Abstraktionsstufen direkt nebeneinander (Abb. 9.31).

Oben das bereits bekannte plattform-neutrale Modell, unten die spezifische Variante, die aus der Transformation hervorgegangen ist. Folgende Vorgehensweise wurde angewandt:

- Betrachtet werden nur `WorkflowSteps`, die Rollen zugeordnet sind, die vom System unterstützt werden. Dies wird angezeigt durch die Markierung der Rolle mit `{system-supports}`.
- `WorkflowSteps` werden „eins zu eins“ in `TaskNodes` transformiert.
- Da Swimlanes in der Welt der Zustandsdiagramme nicht existieren werden die zu den Tasks gehörenden Rollen im Attribut `role` festgehalten.

- Jeder Task wird bei Erzeugung der Arbeitsliste eines konkreten Bearbeiters zugeordnet. Die Ermittlung des Bearbeiters übernimmt ein jBPM-`AssignmentHandler`. Dieser ist „von Hand“ zu erstellen und die resultierende Java-Klasse im Attribut `assignmentHandler` festzuhalten.
- Entscheidungen (im PIM realisiert durch disjunkte Parameter-Sets), die den Outcome zurückgeben, der dem Ergebnis der Aufgabenausführung entspricht, werden im PSM zu disjunkten Transitionen. Hier gilt: Jeder Output-Pin des PSMs, der ein Ausgangsobjekt zurückliefert, wird in eine Transition transformiert. Auch hier gilt also eine „eins zu eins“ Beziehung.

Abb. 9.31 Transformation gPIWfP nach gjBPM



9.2.9 Vom Modell zum lauffähigen Workflow

Was innerhalb der aktuellen Betrachtungen noch fehlt, ist die Betrachtung des endgültigen Resultats unserer Bemühungen: die Beschreibung der Artefakte, die die modellierten Aspekte und gestellten fachlichen Anforderungen in ein ausführbares System überführen. Wir wollen dabei die wesentlichen Stationen des Workflows unserer Anwendung verfolgen und die relevanten Code-Snippets,

die zur Realisierung mit der gewählten Technologie verwendet wurden, angeben.

Ein weiterer wichtiger Punkt wird auch hier sein, herauszustellen, welche Relationen zwischen den bisher betrachteten Modellen und dem hier betrachteten Codemodell bestehen, wie man also aus den bisher erstellten Diagrammen automatisierbar zu den Informationen gelangt, die zu einer generativen Erstellung des Systemrahmens verwendet werden können.

Im Einzelnen werden folgende Aufgabenbereiche der Kontrollflussschicht betrachtet:

- Die Überführung des modellierten Workflows in eine *jBPM Prozessdefinition*.
- Das *Laden der Arbeitsliste* eines Benutzers.
- Das *Starten eines neuen Prozesses*.
- Die *Beendigung einer Aufgabe* und die dadurch bedingte Fortführung eines Prozesses.

Prozessdefinition

Beginnen wir mit der Erstellung der Prozessbeschreibung mit jBPM-Mitteln, die der unterliegenden Workflow-Maschine als Basis zur Überwachung des Kontrollflusses dient. Abb. 9.32 zeigt, wie die textuelle Beschreibung des oben modellierten Workflows in JPDL, der *jBPM Process Definition Language*, ausschaut. Im Folgenden wollen wir einige Schlüsselaspekte der Beschreibung herauspicken und gleichzeitig die Strategie umreißen, mit der diese textuelle Repräsentation aus den zuvor grafisch gefassten Modellen generiert werden kann.

Der Name der Definition in (1) kann direkt aus dem Namen der Zustandsautomaten in Abb. 9.31 übernommen werden (`Antrag_Angebot_Policierung`). Lediglich nicht erlaubte Charakter müssen zuvor in erlaubte Konstrukte umgewandelt werden. Im Beispiel ist dies die Konvertierung der Kommata in Unterstriche. Die eigentliche Definition beginnt mit der Angabe des Startzustands (2), vom dem zwei Transitionen ausgehen: `Angebot anfordern` (3) und `Antrag bestätigen` (4). Auch diese Transitionen bzw. deren Namen sowie die Namen der Zielzustände, zu denen sie führen (4) bzw. (6), können unter Berücksichtigung der in JPDL erlaubten Syntax direkt aus dem PSM entnommen werden.

Abb. 9.32 Prozessdefinition des Beispiels in JPDL

```
<process-definition name='Angebot_Antrag_Policierung'>   (1)
  <start-state>                                          (2)
    <transition name='Angebot_anfordern'                 (3)
                to='Individualangebot_erstellen'         (4)
    />
    <transition name='Antrag_bestätigen'                 (5)
                to='Antrag_pruefen'                      (6)
    />
  </start-state>

<task-node name='Antrag_pruefen'>
    <task name='Antrag_pruefen' blocking='true'>
       <assignment class='genuacs.BeraterPoolA'/>
    </task>
    […]
  </task-node>
  <task-node name='Individualangebot_erstellen'>         (7)
     <task name='Individualangebot_erstellen'            (8)
           blocking='true'>                              (9)
        <assignment class='genuacs.BeraterPoolB'/>      (10)
     </task>
    […]
  </task-node>
  […]
</process-definition>
```

`TaskNodes` des PSMs werden in der Prozessdefinition zu `<task-node>` Elementen, die den Namen des zugehörigen Knotens tragen. Im Falle des Knotens `Individualangebot erstellen` aus dem PSM ist dies das Element mit entsprechend angepasstem Namen (7). In Task-Knoten können prinzipiell mehrere Aufgaben gebündelt werden, in unserem Falle entspricht ein Task-Knoten jedoch einer Aufgabe (bzw. Task), die den gleichen Namen wie der entsprechende Knoten enthält (8). Ebenfalls zu sehen ist, dass der Gesamtprozess erst weitergeführt wird, wenn die Aufgabe als beendet markiert worden ist und es sich somit um eine „blockierende Aufgabe handelt (9).

In (10) schließlich wird der Name der Java-Klasse angegeben, die die Zuordnung von konkreten Task-Instanzen zu dezidierten Bearbeitern übernimmt. Zur Laufzeit passiert Folgendes: Immer wenn die Prozessausführung einen Task-Knoten erreicht, wird eine Instanz der Aufgabe erstellt. Der zugeordnete Assignment-Handler bestimmt einen konkreten Bearbeiter und ordnet die frisch erzeugte Aufgabe seiner Arbeitsliste zu. Ruft der Bearbeiter in Zukunft die

Übersicht über die ihm zugeordneten Aufgaben auf, erscheint diese Aufgabe in der Übersicht. In der ersten Iteration des genua-Frameworks ist der entsprechende Handler per Hand zu schreiben und sein qualifizierter Klassenname in den entsprechenden Aufgabenknoten einzutragen. Von dort aus kann er dann bei der Generierung der Prozessdefinition in Erfahrung gebracht und in die JPDL-Definition eingetragen werden. An dieser Stelle besteht ein Variationspunkt innerhalb der Transformation. In JPDL können AssignmentHandler sowohl pro Task-Knoten, aber auch innerhalb eines `<swimlane>` Tags angegeben werden. Wird die zweite Variante gewählt, gilt ein Handler für alle Knoten innerhalb der entsprechenden Swimlane.

Taskliste eines Benutzers laden

Ist der statische Teil der Prozessdefinition beendet, können wir uns den dynamischen Aspekten des Systems widmen, die sich mit Instanzen der gegebenen Definition beschäftigen. Beginnen wir mit dem ersten Punkt in der Applikation, in der sich ein Bearbeiter die ihm aktuell zugeordneten Aufgaben anzeigen lassen will, also dem Punkt innerhalb der Geschäftslogik, die der Anzeige eines Screens wie in Abb. 9.29 gezeigt unmittelbar vorausgeht.

Bei den Betrachtungen gehen wir davon aus, dass sich der Benutzer bereits am System authentifiziert und so der Anwendung die ID des Benutzers bereits bekannt ist. Um nun die Aufgaben des Benutzers mit der ermittelten ID aus der jBPM-Datenbank zu beziehen sind folgende Schritte notwendig:

- Das *Erzeugen einer jBPM Datenbank-Sitzung* und der *Beginn einer Transaktion* auf dieser.
- Die *Ermittlung der Aufgaben* des Benutzers mit der ID `<ActorID>`.
- *Beenden der Transaktion* und *Schließen der DB-Sitzung*.

Danach können die ermittelten Informationen über die Aufgaben an die GUI übergeben und zur Anzeige der Liste verwendet werden. Auf diesen Teil der Anwendung wollen wir allerdings nicht näher eingehen und uns auf die Betrachtung der jBPM-Portion der Anwendung beschränken. Abb. 9.33 zeigt die Snippets, die die eben aufgeführten Aufgaben für uns erledigen.

Abb. 9.33 Ermittlung der Tasks eines Benutzers

```
// neue jBPM-Session erzeugen & Transaktion starten      (1)
JbpmSessionFactory jbpmSessionFactory =
   JbpmSessionFactory.buildJbpmSessionFactory();
JbpmSession jbpmSession =
   jbpmSessionFactory.openJbpmSession();
jbpmSession.beginTransaction();

// Taskliste für Benutzer laden                          (2)
List taskInstances =
   jbpmSession.
      getTaskMgmtSession().
      findTaskInstances(<ActorId>);

// Transaktion committen …                               (3)
jbpmSession.commitTransaction();
// … und Session schließen
jbpmSession.close();

// Aufgabenliste zusammenbauen, GUI anzeigen […]         (4)
```

In (1) wird die Datenbanksitzung erzeugt und die Transaktion gestartet. Welche Datenbank und welcher Mechanismus zur Persistierung von jBPM verwendet wird, ist an dieser Stelle transparent und kann an anderer Stelle konfiguriert werden. Interessanter für unsere Zwecke ist der Code in (2). Hier wird mithilfe einer Task-Management-Session die Liste aller Aufgaben (`TaskInstances`) ermittelt, die aktuell dem Benutzer mit der ID `<ActorID>` zugewiesen sind. Die zurückgelieferten Instanzobjekte enthalten alle Informationen, die zur Darstellung der Arbeitsliste bzw. zur Ermittlung weiterer Informationen zu den ermittelten Aufgaben benötigt werden (4). In (3) schließlich wird die Transaktion beendet und die Sitzung geschlossen.

Starten eines neuen Prozesses

Außer der Anzeige der ihm zugeordneten Aufgaben in Form seines Eingangskorbs bzw. Arbeitsliste steht dem Mitarbeiter auch die Möglichkeit zur Verfügung einen neuen Prozess zu starten (vgl. Abb. 9.28). Hierunter werden sowohl Einzelaufgaben verstanden (wie die nicht-personenbezogene Ermittlung eines Versicherungsbeitrags für ein bestimmtes Produkt) aber auch mehrschrittige Prozesse, die in ihrem Verlauf Einträgen in Arbeitslisten von Mitarbeitern erzeugen, die den Prozess nicht selbst angetriggert haben.

Technisch gesehen entspricht der Start eines Prozesses der Instanziierung einer zuvor in JPDL spezifizierten Prozessdefinition.

Abb. 9.34 zeigt die Code-Snippets, die bei diesem Vorgang zur Ausführung kommen.

```
// Prozessdefinition laden                                    (1)
ProcessDefinition processDefinition =
   jbpmSession.
      getGraphSession().
      findLatestProcessDefinition("<WF-Name>");

// Prozessinstanz erzeugen                                    (2)
ProcessInstance processInstance =
   new ProcessInstance(processDefinition);

// OK, los geht's! (Prozess starten)                          (4)
Token token = processInstance.getRootToken();
token.signal("<Transitions-Name>");

// Prozess speichern                                          (5)
jbpmSession.getGraphSession().
   saveProcessInstance(processInstance);
```

Abb. 9.34 Starten eines Prozesses

Das Öffnen einer Sitzung und die Anzeige des Transaktionsbeginns geschieht wie gehabt (vgl. Abb. 9.33), weshalb wir sofort mit der Beschreibung der Folgeschritte fortfahren. In (1) wird zuerst die zuvor installierte Prozessdefinition geladen (auf den Deployvorgang selbst wollen wir an dieser Stelle nicht eingehen und auf die jBPM-Dokumentation in [JBPM] verweisen). Die Definition wird durch die Angabe des Namens spezifiziert, der (wie schon bei der Erzeugung der Prozessdefinition selbst) aus dem Namen des Diagramms in Abb. 9.31 ermittelt werden kann (also bspw. `Antrag_Angebot_Policierung`). Sind mehrere Versionen einer Spezifikation desselben Namens installiert, kann hier auch eine spezielle Versionsnummer angegeben werden. Wir beschränken uns darauf die Aktuellste zu beziehen (`findLatest...()`).

Haben wir die Definition des Prozesses in der Hand, können wir uns eine Instanz davon beschaffen. Dies geschieht schon fast profan indem wir uns mit Java-Bordmitteln eine Instanz der Klasse `ProcessInstance` schaffen, und dem Konstruktor der Klasse die zuvor besorgte Definition mitgeben (2). Frisch erzeugt, liegt das imaginäre Kontrollfluss-Token auf dem Startknoten der Prozessinstanz. Um tatsächlich den „Startschuss“ zu geben, müssen wir dem Token nur noch sagen in welche Richtung es losrollen kann, was wir ihm in (4) signalisieren. Existiert vom Startknoten mehr als eine Möglich-

keit (mehrere ausgehende Transitionen), ist der Name der Transition anzugeben, der genommen werden soll.

Hat beispielweise der Kunde online die Erstellung eines Individualangebotes angefordert, erzeugt das System eine Prozessinstanz des Prozesses `Antrag_Angebot_Policierung` und startet ihn unter Angabe der Transition `Angebot_anfordern`. Hierdurch würde im Kontrollfluss der Knoten `Individualangebot erstellen` erreicht (vgl. Abb. 9.31), der wiederum einen Eintrag in der Arbeitsliste des ermittelten Beraters erzeugt.

In (5) wird der aktuelle Stand des Kontrollflusses gespeichert und ist damit sitzungsübergreifend persistiert. Nachfolgend ist auch hier die Transaktion zu committen und die Sitzung zu schließen. Dieser Vorgang wurde bereits in den vorigen Abschnitten erläutert.

Prozess fortführen (Beendigung einer Aufgabe)

Bleibt nur noch eine Station im Lebenszyklus einer Aufgabe zu beschreiben: was passiert, wenn eine Aufgabe (mit oder ohne Erfolg) abgeschlossen wird, und damit der Kontrollfluss weitergeführt wird.

Abb. 9.35 zeigt, wie dies technisch bewerkstelligt wird. Auch hier wird der Start von Sitzung und Transaktion als bereits erledigt vorausgesetzt.

Abb. 9.35 Beendigung einer Aufgabe

```
// Prozess- und Taskinstanz zu TaskID finden                    (1)
TaskInstance taskInstance =
   jbpmSession.
   getTaskMgmtSession().
   loadTaskInstance(<taskInstanceId>);

ProcessInstance processInstance =
   taskInstance.
   getTaskMgmtInstance().
   getProcessInstance();

// Prozess fortführen (Task beenden)                            (2)
taskInstance.end(<Transitions-Name>);
```

Erste Aufgabe ist es, anhand der Task-ID die zugehörige Task-Instanz in der Prozessdatenbank zu finden. Dies geschieht bequem durch Aufruf der entsprechenden Methode, die uns von einer Task-Management-Session zur Verfügung gestellt wird (1). Da wir später auch den aktuellen Stand der Prozessinstanz wieder persistieren müssen, besorgen wir uns auch diese.

In (2) schließlich wird die Aufgabe als beendet markiert und damit der Kontrollfluss des Gesamtprozesses fortgeführt. Dies resultiert entweder in der Erzeugung einer oder mehrerer Aufgaben und damit in Arbeitslisten-Einträgen oder in der Beendigung des Prozesses. Bei der Anzeige der Beendigung kann auch hier wieder das Ergebnis der Erledigung in Form der zu nehmenden Transition angegeben und damit der Kontrollfluss des Prozesses beeinflusst werden. Nach erfolgter Beendigung muss die Prozessinstanz gesichert, die Transaktion commited und die Sitzung geschlossen werden, alles Schritte, die wir bereits kennen und hier nicht wieder aufgeführt sind.

9.3 Geschäftslogik/Services

Kommen wir nun zum „Herzstück“ einer Architektur, der Realisierung der Geschäftslogik- bzw. Serviceschicht. Wie bereits angeklungen, wurde in dieser ersten Iteration des Framework-Projekts die Entscheidung getroffen, Services als Plain-Java-Methoden zu realisieren und diese innerhalb Stateless Session Beans (SLSBs) zu kapseln. Um trotzdem in späteren Iterationen flexibel zur Erweiterung zu bleiben, wurde zusätzlich entschieden die Anbindung der Services an die Präsentationsschicht durch den Einsatz von Beehive Controls zu kapseln.

9.3.1 Beehive Controls

Beehive Controls (BCs) [Beehive] kapseln den Zugriff auf Java EE *Ressourcen*, indem sie stellvertretend für diese die direkte Kommunikation mit der aufrufenden Geschäftslogik übernehmen. Der Geschäftslogik (dem *Client*) präsentieren sie sich als instanziierbare JavaBeans und stellen alle angebotenen Features der Ressource wie Konfigurationsoptionen, Methodenaufruf und Lifecycle-Callbacks unter einer einheitlichen Struktur zur Verfügung, die unabhängig vom tatsächlichen Charakter der unterliegenden Ressource ist.

Im genua-Framework wird dieser Mechanismus dazu verwendet, eine klare Trennung der Präsentations- von der Serviceschicht zu erreichen. Aufrufe der Geschäftslogik geschehen immer über die delegierende Zwischenstelle eines Controlaufrufs. Obwohl in der ersten Iteration des Projektes die Festlegung getroffen wurde, dass Services ausschließlich als POJO-Methoden innerhalb SLSBs realisiert werden, ist durch die konsequente Anwendung des Delegationsmechanismus eine Veränderung der unterliegenden Technologie (z. B. Webservices) in zukünftigen Releases jederzeit möglich.

Abb. 9.36 zeigt die Architektur von Beehive Controls im Überblick. *Clients* repräsentieren den Quellcode der Applikationen, die Zugriff auf die Java EE *Ressourcen* benötigen. Der Zugriff selbst geschieht über ein *Control*, das die Anfragen des Clients an die Ressource delegiert.

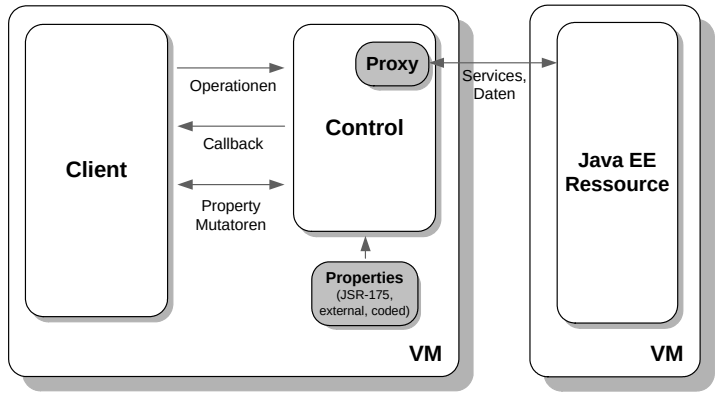


Abb. 9.36 Beehive Controls – Architekturüberblick

Technisch geschieht dies über einen Control-internen Proxy, der eine Referenz auf die Ressource hält (z. B. EJB-Home Stub) und die direkte Kommunikation zwischen den Instanzen für das Control übernimmt. Die technischen Details der Kommunikation werden durch die angebundene Ressource bestimmt. Im Falle Java EE SLSBs werden dies wahrscheinlich RMI-Calls sein. Handelt es sich um einen Web-Service, kommen die für diese Technologie vorgesehenen Protokolle und Mechanismen zum Einsatz.

Die Kommunikation zwischen Client und Control geschieht direkt über den Aufruf der Controlmethoden durch den Clientcode. Dies bedingt, dass sich der Code des Clients und der der verwendeten Controls im gleichen Adressraum bzw. in der selben virtuellen Java Maschine (VM) befinden müssen.

Auch der Zugriff auf Attribute der Ressource werden über das zugeordnete Control abgewickelt. Zusätzlich können Control-eigene Attribute definiert werden und beispielsweise zur Konfiguration des Proxys dienen (JNDI-Namen, …). Zur Verwaltung der Attribute stellt Beehive eine Fülle von Möglichkeiten zur Verfügung: Attribute können gemäß JSR-175 (Annotationen) direkt an den Quellcode der Control angetragen, durch Java-Bordmittel (Mutator-Methoden) realisiert, oder dem Konstruktor der Control übergeben werden. Auch die Bindung an externe Artefakte ist möglich. Beispiel: die Bindung von Attributen an die Deployment-Deskriptoren einer Java EE Ressource.

Letztes hier vorgestelltes Feature von BCs ist die Möglichkeit zur Realisierung von Callbacks, die im Falle spezifizierter Ereignisse einen Aufruf von Quellcode im Client realisieren. Diese Ereignisse finden üblicherweise im Lebenszyklus der Ressource statt (z. B. Instanziierung, Destruktion, …). Die Ressource informiert dann das Control über das Auftreten des Ereignisses und ruft letztendlich die spezifizierte Methode des Clients auf.

Technische Realisierung

BCs bestehen aus jeweils einem Interface sowie (mindestens einer) zugehörigen Implementierung. Die Aufgabenteilung zwischen den beiden geschieht wie erwartet: Das Interface listet alle öffentlich zugänglichen Methoden des Controls auf während die Implementierung eine konkrete Realisierung des dort aufgeführten Verhaltens darstellt. Innerhalb des genua-Frameworks wurde vor allem Gebrauch von den deklarativen Fähigkeiten Beehives gemacht was bedeutet, das die Arbeit mit Beehive im Groben ungefähr so aussieht: Erstellung der Interfaces und Implementierung sowie Kennzeichnung der Klassen und Methoden mit Beehive-spezifischen Annotationen. Am Ende steht dann die Verwendung des mitgelieferten Ant-Tasks, der den von Beehive zur Laufzeit benötigten Gluecode erzeugt.

Abb. 9.37 zeigt ein Beispiel für ein Interface, das als Schnittstelle zu einem Beehive Control vorgesehen ist. `HelloWorld` unterscheidet sich nur durch die Annotation `@ControlInterface` (2) aus dem Fundus der Beehive Controls Standard-API (1) von einem „gewöhnlichen“ POJO-Interface und listet alle öffentlich zugänglichen Operationen der zu kapselnden Ressource auf (3) – hier nur `myMessageToTheWorld()`.

```
// Interface File
package genuacs;
import org.apache.beehive.controls.api.bean.ControlInterface;     (1)
@ControlInterface                                                  (2)
public interface HelloWorld
{
   public String myMessageToTheWorld();                           (3)
}
```

Abb. 9.37 Beispiel eines BC-Interfaces

Im vorliegenden Beispiel wollen wir die Implementierung des Interfaces in Form einer POJO-Klasse realisieren, die die aufzurufende Geschäftslogik direkt enthält. Im weiteren Verlauf werden wir noch eine Möglichkeit kennen lernen, bestehende Java EE Session Beans

zur Realisierung zu verwenden und die zugehörige Control aus diesen zu generieren.

Abb. 9.38 zeigt die POJO-Implementierung unseres Beispielinterfaces. Auch hier wird die Klasse durch eine Annotation gekennzeichnet. Neben dem Java-Schlüsselwort `implements` zeigt `@ControlImplementation` (2) den Charakter der Klasse als Realisierung eines BC-Interfaces an. In der zur Drucklegung vorliegenden Version von Beehive (1.0) besteht zudem die zwingende Vorgabe, dass Implementierungen der Namenskonvention `<Interfacename>`**`Impl`** genügen müssen (3), eine Tatsache, die schon mal einiges an Schweiß kosten kann, wenn man sie nicht sofort in der (ansonsten sehr ausführlichen) Dokumentation findet.

Abb. 9.38 Implementierung der BC als POJO

```
// Implementation File
package genuacs;
import org.apache.beehive.controls.api.bean.*;          (1)

@ControlImplementation(isTransient=true)                (2)

public class HelloWorldImpl implements HelloWorld       (3)
{
   public String myMessageToTheWorld()                  (4)
   {
      return "Hello World!";
   }
}
```

Letztendlich findet sich in (4) noch die Geschäftslogik, die die im Interface angegebene Signatur mit Leben füllt. An dieser Stelle ist dies die Rückgabe einer freundlichen Begrüßung.

Schauen wir uns nun an, wie wir den nun in Form eines BC-Interfaces definierten und durch eine BC-Implementierung realisierten Service innerhalb eines Beehive Pageflows, der in genua zur Realisierung von Dialogen realisiert wird, aufgerufen werden kann. Pageflows wurden in Kapitel 8 ausführlich besprochen, sodass wir uns an dieser Stelle ganz auf die Einbindung der BC konzentrieren können.

Abb. 9.39 zeigt, wie die Einbindung technisch realisiert wird. Der Aufruf von Geschäftslogik findet in Pageflow-Controllerklassen (1) innerhalb von `Actions` statt, die besonders gekennzeichnete Java-Methoden darstellen. Soll nun innerhalb einer solchen `Action` ein Zugriff auf eine Ressource geschehen, die durch eine Beehive Control gekapselt ist, kann dies deklarativ durch die Verwendung einer

weiteren Annotation geschehen. Wird ein Attribut vor einer `Action` mit dem Schlüsselwort `@Control` versehen, wird dieses vor der Ausführung der Action automatisch vom Pageflow-Controller mit der Referenz auf die zugehörige BC initialisiert.

```
// Verwendung von Controls in PageflowControllern
// […] Imports
import org.apache.beehive.controls.api.bean.Control;
import genuacs.HelloWorld;

@Jpf.Controller([…])                                          (1)
public class Controller
extends PageFlowController {

@Control private HelloWorld _helloWorldControl;               (2)
@Jpf.Action(
   forwards={@Jpf.Forward(name="success", path="index.jsp")}
)
protected Forward begin() throws Exception
{
   Forward f = new Forward("success");
   // do the business logic stuff here
   String message =                                           (3)
      _helloWorldControl.myMessageToTheWorld();
   f.addActionOutput("myMessage", message);
   return f;
}

// […] Callbacks
```

Abb. 9.39 Aufruf einer BC aus einem Pageflow-Controller

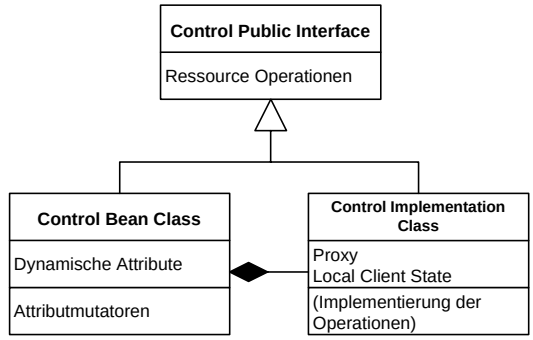
In unserer `Action begin()` bedeutet dies, dass `_helloWorldControl` zur Laufzeit des Systems vor der Ausführung der Methode eine Referenz auf die BC erhält, die die Implementierung kapselt, die wir zuvor angegeben haben, und uns so die dort realisierte Geschäftslogik zur Verfügung stellt (2). Der Aufruf in (3) schließlich geschieht auf dem Interface des Services. Die Tatsache, dass sich hinter diesem Aufruf eine delegierende Instanz befindet (das Control), bleibt dem Client verborgen.

Nuts and Bolts

Zur Realisierung der Ressourcentransparenz und des Delegationsmusters bedienen sich BCs der in Abb. 9.40 dargestellten, technischen Architektur.

Vom Entwickler vorzugeben ist auf jeden Fall das öffentliche *Interface* (im oberen Teil der Abbildung), das die Operationen der zu kapselnden Ressource spezifiziert, die von der Control gekapselt werden sollen. Die *Bean-Klasse* ist der Teil des Musters, mit dem die Clients des Musters in direkte Interaktion treten. Sie übernimmt die Kommunikation mit den Klienten und leitet die Anfragen an die gewrappte *Implementierungsklasse* weiter.

Abb. 9.40 Beehive – Technische Architektur



Das Interface wird immer per Hand erstellt während die Bean-Klasse (und weitere Helper) durch Anwendung eines mitgelieferten Ant-Tasks automatisch generiert wird. Für die Implementierung bestehen prinzipiell zwei Möglichkeiten. Im vorigen Beispiel haben wir die Basisimplementierung selbst erstellt. Im Folgenden werden wir sehen, dass als implementierende Logik auch bereits bestehende Services wie bspw. EJBs verwendet werden können. In diesem Fall ist nur das Interface zu erstellen und durch spezielle Annotationen zusätzliche Information über den bestehenden Service anzugeben. die Anbindung und die Erstellung des Proxys geschieht dann wiederum vollautomatisch durch Anwendung des bereits gennanten Tasks.

Innerhalb des genua-Frameworks haben wir uns für die Verwendung von in Session Beans gekapselten Services entschieden. Der folgende Abschnitt gibt einen Überblick über die zur Anwendung gekommene Architektur der Serviceschicht sowie die technische Realisierung der Anbindung an die Präsentationsschicht unter Zuhilfename von Beehive Controls.

9.3.2 Realisierung der Services als SLSBs

Aufrufe von Services geschehen ausschließlich aus den unterschiedlichen Controllern, die auch den Kontrollfluss der Anwendung auf Dialog- und Workflow-Ebene steuern. Im Falle des Dialogflusses

beispielsweise repräsentieren «SystemCall» Aktionen einen solchen Aufruf.

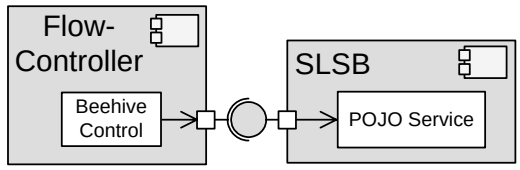


Abb. 9.41 Überblick Servicerealisierung

Auf technischer Seite realisieren einfache Java-Methoden die Realisierung der Geschäftslogik. Querschnittsaufgaben wie Security, transaktionale Aspekte und verteilte Ausführung übernehmen Java EE Stateless Session Beans (SLSBs), in die die POJO-Services gewrappt werden.

Abb. 9.41 verdeutlicht diesen Sachverhalt noch einmal schematisch. Links dargestellt stellvertretend für die unterschiedlichen Ausprägungen die konzeptuelle Komponenten eines `FlowControllers`, in dem sich zur Trennung der Schichten Beehive Controls befinden, die den Zugriff auf die SLSBs kapseln (vgl. voriger Abschnitt).

Beehive System Controls

Abschließend zu diesen Betrachtungen der Serviceschicht wollen wir noch auf ein weiteres Feature von Beehive eingehen, das die Arbeit mit SLSBs als Realisierung von Control-Interfaces wesentlich erleichtert: die *Beehive System Controls* (BSCs) für EJBs.

```
// […] Imports

@ControlExtension                                              (1)
@EJBHome(jndiName="HelloWorldRemote")                          (2)
@JNDIContextEnv(                                               (3)
   contextFactory="org.jnp.interfaces.NamingContextFactory",
   providerURL="localhost:1099"
)

public interface HelloWorld
   extends SessionEJBControl,                                  (4)
           HelloWorldRemote,                                   (5)
           HelloWorldHomeRemote{                               (6)
}
```

Abb. 9.42 Sys.Control des `HelloWorld` Interfaces

Abb. 9.42 zeigt ein BSC im Einsatz. Die Implementierung des Interfaces `HelloWorld`, das wir bereits aus dem vorigen Abschnitt

kennen, wird diesmal durch eine SLSB realisiert. Wesentlich sind folgende Punkte:

- Das Interface wird als `@ControlExtension` gekennzeichnet (1),
- `HelloWorld` erbt von `SessionEJBControl` (Beehive interne Struktur) sowie von den Remote- und Homeinterfaces der unterliegenden SLSB (5+6). Obwohl hier nicht explizit dargestellt, gehen wir dabei davon aus, dass im Remoteinterface die Methode `myMessageToTheWorld()` deklariert ist, die wir bereits aus dem Beispiel im vorigen Abschnitt kennen.

Um die Anbindung abzuschließen, müssen nun nur noch der JNDI-Name unter dem die SLSB deployt ist (2) sowie etwaige Details zur Auflösung dieser Referenz (3) angegeben werden. In unserem Fall sind dies `HelloWorldRemote` als Name und die Zugriffsdaten für ein Standard-Deployment auf JBoss. Aus Platzgründen wollen wir davon ausgehen, dass der Leser mit dem grundsätzlichen Aufbau und den notwendigen Artefakten von Session Beans vertraut ist und deshalb nicht weiter auf Details eingehen.

Ein Aufruf des Beehive Ant-Tasks generiert aus dem vorliegenden Interface alle weiteren Artefakte, die zum Zugriff auf die spezifizierte Bean benötigt werden. Der Zugriff auf das Control aus der Präsentationsschicht heraus unterscheidet sich nicht von dem des vorigen Beispiels (vgl. Abb. 9.39).

9.4 Persistenz

Als letzte noch zu meisternde Schicht der genua-Architektur wurde im Projekt die Entwicklung eines Persistenzkonzepts in Angriff genommen. Auch hier wurde ein plattform-unabhängiges Profil sowie eine prototypische plattform-spezifische Realisierung erstellt.

Innerhalb der Projektplanung (vgl. voriges Kapitel) haben wir uns für Hibernate als technologische Plattform entschieden. Es bleiben folgende Aufgaben zu erledigen:

- Erstellung eines plattform-unabhängigen Profils zur Modellierung des Persistenzaspekts sowie eines plattform-spezifischen Profils auf Basis der Hibernate Plattform.
- Ebenfalls anzugeben ist das Mapping von einem Abstraktionslevel auf das nächste, um als Grundlage zur Erstellung der Transformationsbeschreibungen dienen zu können. Wir wollen diese Beschreibung jeweils im technologiespezifischen Profil

liefern, indem angegeben wird, in welchem Verhältnis die dort definierten Elemente zu ihren plattform-unabhängigen Pendants stehen.

9.4.1 genua Platform Independent Persistence Profile (gPIP)

Für die plattform-unabhängige Modellierung einfacher Persistenz-Aspekte wurde auf einem Ansatz von Scott W. Ambler aufgebaut [Amb04]. Allerdings wurde in der ersten Version nur eine kleiner Teil der dort vorgestellten Modellelemente verwendet und zum Teil neue Elemente hinzugefügt.

Als Literaturverweis aufgeführt ist die Online-Version des Basisprofils von Ambler. Als zusätzliche Lektüre des gleichen Autors empfiehlt sich dessen Buch *Agile Database Techniques* [Amb03], das außer einer ausführlichen Beschreibung des Profils in die Konzepte und Best-Practices zur Erstellung von Datenmodellen in agilen Software-Projekten sowie in grundsätzliche Überlegungen zur Schaffung eines solchen Profils einführt.

Das Ergebnis dieser Bemühungen wird am Ende dieses Abschnittes das *genua Platform Independent Persistence Profile (gPIP)* sein, das dann im weiteren Verlauf des Projektes zur Erstellung von PIMs genutzt werden kann. Abb. 9.43 zeigt die Definition des Profils mittels UML.

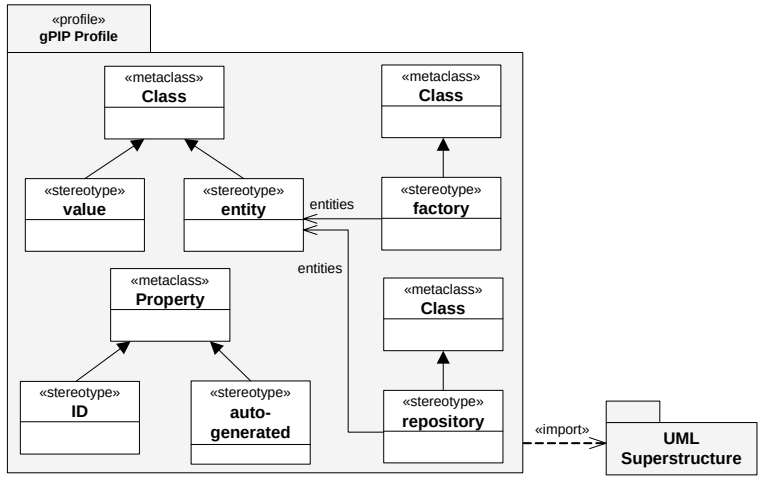


Abb. 9.43 Definition des gPIP

Das Profil besteht aus insgesamt sechs Stereotypen, die sich in ihrer Anwendbarkeit in drei Bereiche einteilen lassen. Zum einen die Ste-

reotypen «entity» und «value», die auf *Klassen* sowie «ID» und «auto-generated», die auf *Attribute einer Klasse* angewandt werden könne. Zum anderen «factory» und «repository», die die Services zum Erzeugen, Finden, Ändern und Löschen (CRUD [CRUD]) zur Verfügung stellen.

«entity»

Mit «entity» gekennzeichnete Klassen stellen Entitäten dar, die eine *Identität* besitzen, die über die aktuelle Wertbelegung ihrer Attribute hinausgeht. So sind zwei Entitäten selbst dann verschieden, wenn alle ihre Attribute die gleichen Werte besitzen. Weiterhin besitzen Entitäten persistenten Charakter. Veränderungen an ihren Werten werden entweder automatisiert am Ende einer Sitzung oder (wie in der aktuellen Iteration des Frameworks) durch manuelles Anstoßen in die unterliegende Datenbank übernommen.

Attribute einer Entität können mit folgendem Stereotypen weiter detailliert werden:

- **«ID» (mandatorisch)** → Kennzeichnet ein Attribut als Teil der *Identifikation* der Entität. Die Menge aller mit «ID» gekennzeichneten Attribute ergibt ein unter allen gleichartigen Entitäten eindeutiges und einmaliges Identifizierungsmerkmal, das die Identität des Objektes bestimmt. Der Wert einer Identifikation kann automatisch bei Erzeugung einer Entitätsinstanz von der Applikation bzw. der Datenbank generiert oder manuell gesetzt werden. Soll diese Aufgabe automatisiert erledigt werden, kann man dies durch die Verwendung des Stereotypen «auto-generated» modellieren.

«value»

Im Gegensatz zu Entitäten besitzen «value»-Klassen keine eigene Identität und damit auch keine mandatorische Identifikation. Sie dienen lediglich als Container für Daten, die zwischen anderen Klassen ausgetauscht werden. «value»-Klassen gehören immer eindeutig zu einer Entität und werden über eine Kompositionsbeziehung an diese gebunden.

«factory»

Eine Factory stellt höher gelegenen Schichten den Service zur Erzeugung von Entitäten und deren zugeordneten Value-Objekten zur Verfügung. Dabei kann ein Objekt oder ein komplettes Objektnetz

erzeugt werden. Per Assoziation (`entities`) kann festgelegt werden, für welche Entitäten eine Factory „verantwortlich“ ist.

«repository»

Zeichnen sich Factorys für die Erzeugung von Entitäten verantwortlich, stellen Repositorys die Funktionalität zum Suchen, Ändern und Löschen von Entitäten zur Verfügung. Auch hier wird per Assoziation modelliert, welches Repository für welche Entität zuständig ist.

Domain Driven Design (DDD)

Das Konzept der `entity/value` Objekte sowie das Factory-/Repository-Konzept wurde dem Buch *Domain Driven Design* von Eric Evans [Eva03] entnommen. Für eine ausführlichere Beschreibung des Konzepts möchten wir auf das „Original“ verweisen und mit der Beschreibung fortfahren.

Anwendungsbeispiel

Abb. 9.44 zeigt das gPIP im Einsatz: ein `Angestellter` wird eindeutig durch seine Personalnummer (`PNummer`) identifiziert, die im Unternehmen jeweils nur einmal vergeben wird.

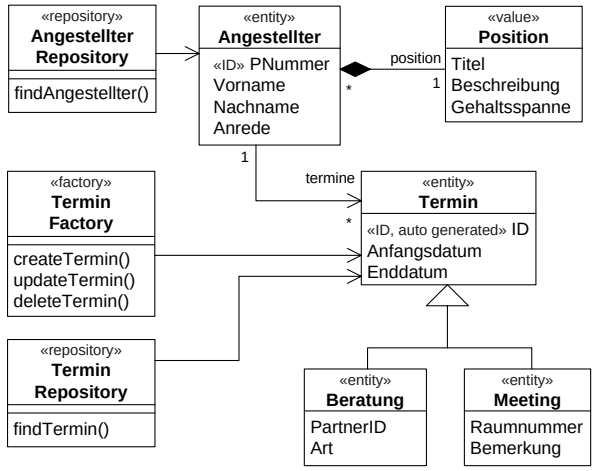


Abb. 9.44 Beispiel für den Einsatz des gPIP

Die Position, die er innehat, ist als `«value»`-Objekt über eine Kompositionsbeziehung mit ihm verbunden. Dies bedeutet, dass dem Programmierer im späteren Quellcode zwar selbstständige `Positions`-Objekte (/-Klassen) zur Verfügung stehen, deren Daten jedoch datenbankseitig wahrscheinlich in derselben Tabelle – und damit jeweils im selben Datensatz – gespeichert werden, wie die des korrespondierenden Angestellten. Die Verwendung der Einschränkung *„wahrscheinlich“* ergibt sich daraus, dass an dieser Stelle

(plattform-unabhängiges Modell) die technische Realisierung und damit auch das Objekt/Relationale Mapping der Attribute auf die konkreten Datenbank-Strukturen, noch nicht bekannt ist.

Termin-Entitäten besitzen eine auto-generierte `ID` sowie eine gerichtete 1-zu-n Beziehung zum korrespondierenden `Angestellten`, der seine zugeordneten Termine über die gleichnamige Assoziation erreichen kann. Sowohl `Meeting` als auch `Beratung` stellen Spezialisierungen von Terminen dar.

Spätestens an dieser Stelle wird die Unabhängigkeit der Modellierung von der technischen Realisierung deutlich. Relationale Datenbanksysteme (RDBMS) kennen das Konzept der Generalisierung (Vererbung), das aus dem Bereich der Objektorientierung stammt, nicht. Jedoch existieren Mechanismen, um sowohl Generalisierung als auch das eng damit gekoppelte Konzept der Polymorphie für den Benutzer transparent zu simulieren. Vorausgesetzt die Realisierung soll auf RDMS-Basis geschehen, ist für das aus diesem Modell resultierende plattform-abhängige Modell also eine Technologie (Plattform) zu wählen, die dies leisten kann. Im Falle von genua ist dies mit der Auswahl des des Hibernate Frameworks geschehen, das die Vererbung unter Entitätsklassen erlaubt und in die relationale Welt abbildet.

Neben den Entitäts- und Value-Klassen finden sich noch die zugehörigen Factorys und Repositorys am rechten Rand der Abbildung. Während für Termine beide Arten von Service zur Verfügung stehen, existiert für Angestellte nur ein Repository, da im Beispiel nur bereits existierende Instanzen gesucht und nicht neu erzeugt werden sollen.

Der folgende Abschnitt definiert das Profil für die Modellierung ausgewählter Hibernate-Funktionalität mittels UML und erläutert gleichzeitig das Mapping der in diesem Abschnitt als plattformunabhängig definierten Konzepte auf die verwendete Technologie. Die Beschreibung dieses Mappings dient dann im weiteren Projektverlauf als Basis zur Definition der Transformationsbeschreibungen.

9.4.2 genua Hibernate Persistence Profile (gHPP)

Schauen wir uns nun den Prototyp eines Profils an, der die in Abschnitt 9.4.1 erläuterten Persistenz-Konzepte auf die in Hibernate verfügbaren technologischen Mittel abbildet.

Anwendungsbeispiel

Bevor wir das komplette Profil formal angeben, wollen wir zuerst schauen, wie das im vorigen Abschnitt vorgestellte Beispiel (Abb. 9.44) in gHPP dargestellt werden kann.

Der erste Teil des Modells (Klasse `Angestellter` und Kompositionsbeziehung zu `Termin`) wird folgendermaßen realisiert (siehe Abb. 9.45):

- *Entitäten* (Stereotyp `«entity»`) werden auf *Hibernate-Entitäten* gemappt (Stereotyp `«hibernate-entity»`)
- während `«value»`-Klassen durch *Hibernate-Komponenten* realisiert werden (`«hibernate-component»`).

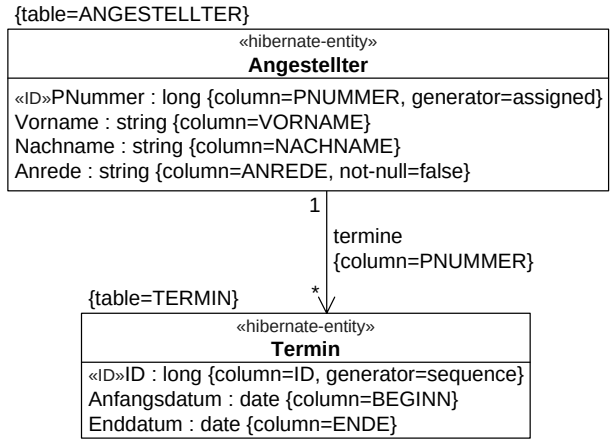


Abb. 9.45 gHPP Beispiel – Teil I

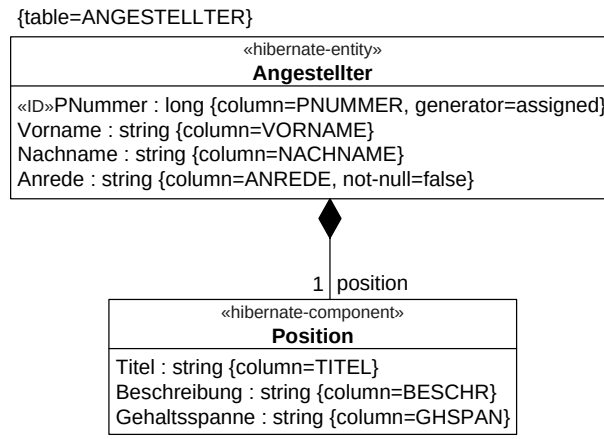
Zusatzinformationen, die zur Generierung der entsprechenden Artefakte benötigt werden, werden sowohl in Form von Stereotypen, als auch als *Tagged-Values* an die betroffenen Elemente angetragen. Da die meisten Annotationen des Profils intuitiv verständlich sind, wird für die genaue Definition auf die Referenz am Ende dieses Abschnittes verwiesen.

An dieser Stelle wollen wir nur kurz auf das Mapping der Identifikation `Angestellter::PNummer` eingehen. Identifikationen entsprechen auch in Hibernate Terminologie IDs und werden folgerichtig mit dem Stereotypen `«ID»` versehen. Hibernate bietet eine Vielzahl an Möglichkeiten diese Identifikationen zu generieren. Im Beispiel (Abb. 9.45) haben wir uns dafür entschieden, bei der Erzeugung eines neuen `Angestellter`-Objekts den Wert des Primärschlüssels manuell zu setzen, was durch das *Fehlen* des Stereo-

typen «auto-generated» am Attribut PNummer modelliert ist. Hibernate bietet diese Möglichkeit durch Angabe von assigned als ID-Generator-Strategie und wird von gHPP durch die Angabe des entsprechenden Bezeichners im generator-Wert ermöglicht.

Den zweiten Teil unseres Beispiels (gemappt auf die Hibernate-Plattform) stellt Abb. 9.46 dar. Beachten Sie bitte, dass Position weder die Angabe einer ID, noch eines zugehörigen Tabellennamens angehängt sind. Dies ergibt sich aus dem «value»-Charakter der Klasse (vgl. Abschnitt 9.4.1).

Abb. 9.46 gHPP Beispiel – Teil II



gHPP Definition

Nach diesem einleitenden Beispiel wollen wir die prototypische Definition des Profils angeben, wie sie in der ersten Iteration des Projekts entstanden ist. Abb. 9.47 zeigt die UML-Version. Danach werden die einzelnen Stereotypen genauer in textueller Form beschrieben.

«hibernate-entity»

Mit «hibernate-entity» versehene Klassen stellen die Realisierung von *Entitäten* dar, wie sie im Abschnitt über die plattformunabhängige Modellierung von Persistenzaspekten mittels gPIP (Abschnitt 9.4.1) definiert worden sind.

«hibernate-component»

Hibernate-Komponenten realisieren dementsprechend das Konzept der *value-Klassen*, das im selben Abschnitt behandelt wurde, und auf das an dieser Stelle nicht noch einmal eingegangen werden soll.

«ID»

Die Identität von Hibernate-Entitäten wird durch den `«ID»`-Stereotyp realisiert.

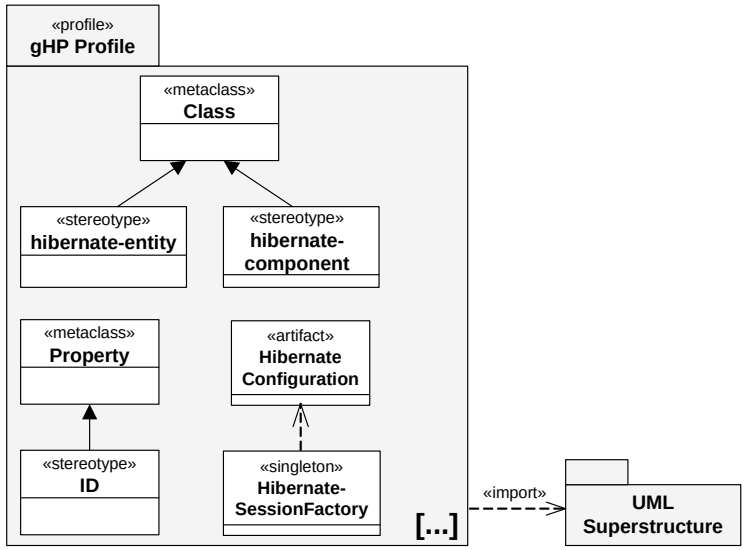


Abb. 9.47 Definition des gHPP

Zusätzlich existieren Tagged-Values, die das mittels gHPP erstellte Modell weiter detaillieren können. Eine Übersicht über deren Anwendbarkeit und Bedeutung liefert Abb. 9.48.

Tagged Value	Anwendbar auf	Bedeutung
column	alle Attribute	Gibt an, auf welche konkrete Tabellenspalte der unterliegenden relationalen Datenbank das Attribut gemappt werden soll
generator	«ID»-Attribute	Gibt die Generierungs-strategie für Primär-schlüssel (IDs) an, die von Hibernate bei der Erzeugung neuer Objekte angewandt werden soll
not-null	alle Attribute	Standardmäßig auf `false` gesetzt gibt dieses Flag an, ob DB-seitig ein `not-null` Constraint auf die entsprechende Spalte gelegt werden soll. Für alle `ID`s gilt: `not-null == true`.
table	Hibernate-Entitäten («hibernate-entity»)	Gibt die Tabelle an, auf die eine Entität (Klasse) gemappt werden soll.

Abb. 9.48 Tagged-Values Übersicht (Ausschnitt)

«HibernateSessionFactory»

Hierbei handelt es sich um ein Singleton zur Erzeugung von Hibernate Sessions, das später von den Repository- und Factory-Klassen verwendet werden kann. Konfigurative Details können der Factory über das Artefakt `HibernateConfiguration` mitgeteilt werden. In Abschnitt 9.4.4 werden wir auf die Verwendung der Factory im Zusammenspiel mit den übrigen Komponenten des genua-Frameworks noch etwas näher eingehen.

9.4.3 Strategie zur Abbildung Hibernate → gHPP

Angestrebt ist die transformatorische Abbildung aller Hibernate-Funktionalitäten auf das gHPP. Betrachten wir dazu noch einmal den ersten Teil des Beispiels (Abb. 9.45) und vergleichen diesen mit den Mapping-Dateien, die ein Generator aus diesen Informationen erstellen könnte (Abb. 9.49, Abb. 9.50).

Abb. 9.49 Mapping der Klasse Angestellter

```
<class name = "Angestellter" table = "ANGESTELLTER">
   <id name = "pNummer" column = "PNUMMER" type = "long">
     <generator class = "assigned"/>
   </id>
   <property name = "vorname"
       type = "string"
       column = "VORNAME"
       notnull = "true" />
              […]
   <set name = "termine">
      <key column = "PNUMMER"/>
      <one-to-many class = "Termin"
   </set>
</class>
```

Abb. 9.50 Mapping der Klasse Termin

```
<class name = "Termin" table = "TERMIN">
   <id name = "id" column = "ID" type = "long">
     <generator class = "sequence"/>
   </id>
   <property name = "anfangsdatum"
       type = "date"
       column = "BEGINN"
       notnull = "true" />
   <property name = "enddatum"
       type = "date"
       column = "ENDE"
       notnull = "true" />
</class>
```

Sehen wir die Klassen (entnommen aus Abb. 9.45) als Eingabe und die angegebenen Hibernate-Mappings als zu realisierende Ausgabe, ist folgendes Vorgehen denkbar:

Konfigurationseinstellungen der Mapping-Datei, die durch eigene Elemente dargestellt werden (`<class>`, `<id>`, `<set>`, …), sollen so weit wie möglich durch (stereotypisierte) UML-Konstrukte, wie Klassensymbol, Assoziation, Attribut usw. dargestellt werden. Können diese in Hibernate weiter konfiguriert werden (*Attribute* innerhalb der Elemente) werden diese Attribute als *Tagged-Values* an das korrespondierende UML-Element angehängt.

Da harter Code in diesem Falle tatsächlich einmal mehr sagt als tausend Worte, hier noch der zweite Teil des Beispiels (Abb. 9.46) mit den zugehörigen Mappings nach oben erläutertem Vorgehen.

```
<class name = "Angestellter" table = "ANGESTELLTER">
   <id name = "pNummer" column = "PNUMMER" type = "long">
     <generator class = "assigned"/>
   </id>
   <property name = "vorname"
       type = "String"
       column = "VORNAME"
       notnull = "true" />
              […]
   <component name = "position" class = "Position">
       <property name = "titel"
          type = "String"
          column = "TITEL"
          notnull = "true" />
              […]
   </component>
</class>
```

Abb. 9.51 Mapping der Klasse Angestellter

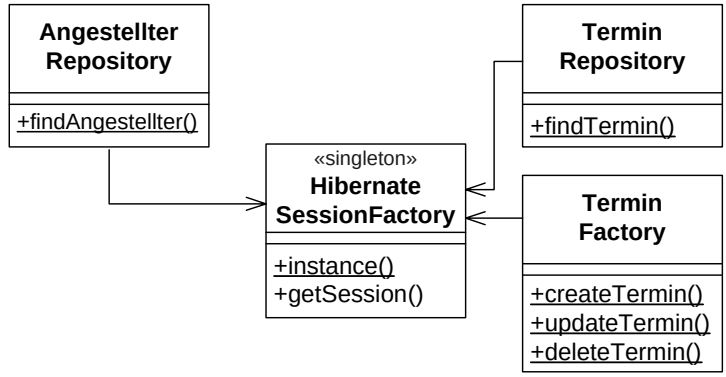
Bitte beachten Sie, dass zur Veranschaulichung die beiden Assoziationen der Klasse `Angestellter` getrennt bei der Angabe der Mappings behandelt wurden. Im Normalfall müssten die Ergebnisse aus Abb. 9.49 und Abb. 9.51 in einer gemeinsamen Datei zusammengeführt werden. Da dieser Vorgang jedoch trivial ist und keine weiteren Einsichten liefert, wollen wir auf die Angabe dieser Vereinheitlichung verzichten und mit der Beschreibung des Zusammenspiels der Einzelkomponenten im Framework fortfahren.

9.4.4 Anbindung an die Serviceschicht

Abb. 9.52 zeigt, wie die Services der Persistenzschicht den höher gelegenen Schichten zur Verfügung gestellt werden. Generiert werden insbesondere folgende Artefakte:

- Basierend auf den Diagrammen des PSMs die Repository-/Factoryklassen, die die Services kapseln.
- Hibernate-Utility-Klassen, die von den generierten Persistenzservices verwendet werden. In der Abbildung ist dies lediglich die Hibernate Session-Factory.

Abb. 9.52 Services der Persistenzschicht



Die Verwendung der generierten Klassen geschieht in der ersten Version des Frameworks noch rein manuell. Sollen Entitäten persistiert, geladen oder gelöscht werden, wird dies innerhalb der POJO-Implementierung der Serviceschicht direkt über die Verwendung der entsprechenden Repository-/Factoryklassen erledigt.

Wie man ebenfalls in der Abbildung sieht, geschieht die Realisierung dieser Klassen momentan durch die Verwendung statischer Methoden. Da dies jedoch Nachteile (bspw. im Bereich des Testens) mit sich bringt, wird überlegt in späteren Versionen dieses Vorgehen durch die Verwendung von IoC-Mustern (Inversion of Control) zu ersetzen. Eine Einführung in den Themenkomplex im Kontext Domain Driven Design bietet [Ric06], auf das wir den interessierten Leser verweisen wollen. Eine ausführlichere Diskussion, obwohl sicherlich berechtigt, notwendig und hochinteressant, muss an dieser Stelle unterbleiben.

9.5 genua Model2Model-Transformator (gM2M)

Der Modelltransformator sowie die dazugehörige Transformationsbeschreibungssprache *gATL*, bilden neben den bereits erläuterten UML-Profilen die Kernkomponenten des genua-Frameworks. Im vorliegenden Abschnitt wollen wir uns den Ablauf einer Model-zu-Model-Transformation sowie deren konkrete Realisation anschauen, und im Detail die bei der Implementierung beteiligten Technologien vorstellen

9.5.1 Essenzieller Ablauf

Bevor wir uns an die technischen Details wagen, wollen wir uns zuerst einmal die grundsätzlichen Schritte, die während einer Transformation zwischen Modellen durchzuführen sind, vor Augen halten. Abb. 9.53 verdeutlicht die grundsätzliche Vorgehensweise, die auch im Kapitel über Transformationen (Kapitel 5) bereits erläutert worden ist.

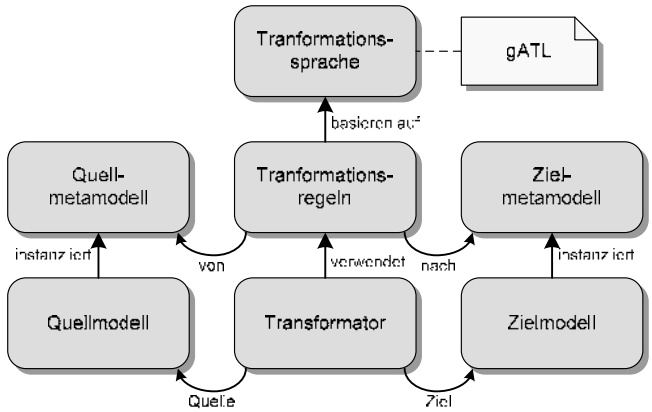


Abb. 9.53 Transformation gM2M (nach [BHL+04])

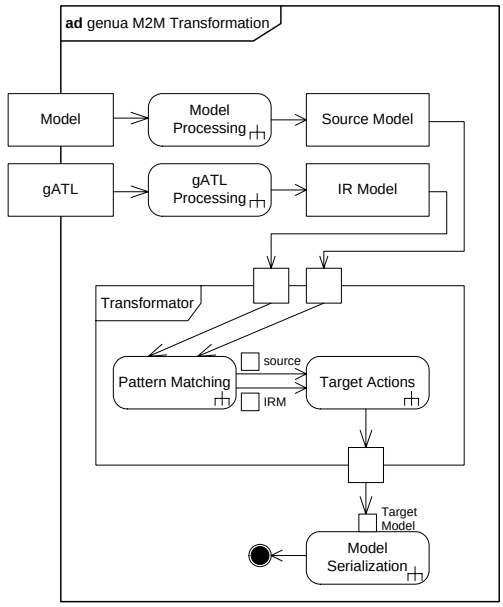
Ein Transformator generiert ein Zielmodell, indem er Elemente aus dem Quellmodell auswählt (deklarativer Teil der Transformationsbeschreibung) und dann Aktionen auf diesen Elementen ausführt (der entsprechende imperative Teil), wie zum Beispiel die Instanziierung von Elementen des Zielmodells. Die Beschreibung geschieht dabei auf Grundlage der unterliegenden Metamodelle und findet im M&M-Projekt durch die Anwendung der *genua Transformation Language (gATL)* statt.

Bevor wir auf Einzelheiten der Beschreibungssprache eingehen (Abschnitt 9.5.2), wollen wir uns zuerst einen Überblick über die einzel-

nen Schritte verschaffen, die vom *gM2M* innerhalb einer Transformation durchgeführt werden. Abb. 9.54 stellt den grundsätzlichen Ablauf in Form eines Aktivitätsdiagrammes dar.

Eingabe in den Transformationsvorgang sind das serialisierte Quellmodell sowie das gATL-Transformationsmodul, in dem die Regeln nach denen die Transformation stattzufinden hat, in Textform abgelegt sind. Wie zu sehen ist, werden im ersten Schritt des Transformationsvorgangs beide Eingaben parallel in Objektbäume deserialisiert. Auf diesen finden dann alle weiteren Bearbeitungen statt.

Abb. 9.54 genua M2M Transformator



9.5.2 genua ATLAS Transformation Language (gATL)

Wie wir bereits in Kapitel 5 erfahren haben, versucht die OMG mit der QVT zur Zeit einen Standard für eine Transformationssprache zu etablieren. Der Reifegrad dieses Standards, war jedoch zu Beginn von M&M-online von einem finalen, anwendbaren Resultat noch weit entfernt. Da nichtsdestotrotz in genua Transformationen irgendwie beschrieben werden müssen, wurde vom Domain-Engineering Team entschieden eine Sprache zu adaptieren, die von der Universität Nantes im Rahmen des *ATLAS*-Projektes entstanden ist, und unter dem Namen *ATL (Atlas Transformation Language)*

[ATL] auch als Antwort auf den QVT-RFP an die OMG eingereicht worden ist und auch in der aktuellen Version des Standards erhebliche Spuren hinterlassen hat.

Grundlagen

Bei ATL handelt es sich um eine hybride Sprache, die sowohl deklarative Konstrukte – zur Selektion von Modellelementen – als auch imperative Sprachanteile enthält, die zur Spezifizierung der Aktionen dienen, die auf den gefundenen Elementen durchgeführt werden sollen. Schauen wir uns ein Beispiel für eine Transformationsbeschreibung in gATL an (Abb. 9.55).

Übersetzungseinheiten ÜE (engl. transformation module) stellen eine Sammlung von Transformationsregeln dar und werden über ihren Namen identifiziert (1). Nach der Angabe dieses (eindeutigen) Modulnamens muss weiterhin spezifiziert werden, welche *Metamodelle* (vgl. Abb. 9.53) an der Transformation beteiligt sind (2). Einmal getroffen gilt diese Festlegung dann für alle folgenden Transformationsregeln des Moduls, wobei `IN` das Quellmetamodell, und `OUT` das Zielmetamodell spezifiziert.

Als Nächstes folgt die Angabe der einzelnen *Transformationsregeln (engl. rules)* (3). Hier werden nach dem im Modul eindeutigen Namen der Regel zuerst die Deklarationen der Quellelemente nebst zugehöriger Bedingungen angegeben, die die Kandidaten erfüllen müssen, um an der Transformation teilnehmen zu können (4). Die Deklaration eines Quellelementes erfolgt dabei nach dem Schema `Bezeichner:Namensraum!Typ`. Nach dieser Deklaration des Typs können nachfolgend in runden Klammern zusätzliche Bedingungen angegeben werden, die zur Teilnahme des Elements an der Transformation erfüllt sein müssen.

Zusammengefasst ergeben sich also die folgenden Bedingungen, die ein Modellelement erfüllen muss, um an der Transformation teilnehmen zu können:

- Es gehört zu einem Namensraum, der im zuvor angegebenen Quellmetamodell `IN` liegt,
- es hat den verlangten `Typ`
- und es erfüllt die zusätzlich gestellten Bedingungen.

Wie bereits erwähnt, ist die Anzahl der neben dem Typ an die Elemente gestellten Zusatzbedingungen beliebig. Zu beachten ist lediglich, dass diese mit Semikola getrennt werden und die einzelnen

Ausdrücke zu `WAHR` oder `FALSCH` (`true` bzw. `false`) auswertbar sein müssen.

Mit (5) folgt nun der imperative Teil, der die Aktionen beschreibt, die vom Transformator auf den Elementen ausgeführt werden, die alle Anforderungen aus (4) erfüllen. Dabei gilt folgende Reihenfolge der Aktionen: Zuerst wird eine Instanz des unter „`to`“ spezifizierten Zielelementes geschaffen und dann die Aktionen im folgenden Klammerblock ausgeführt. Die Angabe der zu erzeugenden Zielelemente/Aktionskombinationen ist nicht auf eins beschränkt. Im Beispiel wird zusätzlich zu einem Element des Typs `JAVA!Class`

Abb. 9.55 gATL Beispiel

```
// module identifier                                          (1)
module UML2JavaExample;

// specification of metamodels                                (2)
create OUT: Java from IN: UML2;

// rule description                                           (3)
rule UMLClass2JavaClass {
      // source and conditions                               (4)
      from c : UML2!Class (
            c.name = 'ExampleClass';
      )
      // target actions                                      (5)
      to j : JAVA!Class (
            j.name <- c.name;
            // Jython block                                  (6)
            JY:
                  attribs = c.getOwnedAttributes()
                  for a of attribs
                        j.getOwnedAttributes().add(a)
            YJ;
      )
      t : GENUA!Tracker (
            t.description = 'UML Class to Java Class';
      )
}
```

ein Element des Typs `GENUA!Tracker` instanziiert, das später die Rückverfolgung der Transformation erlaubt. So werden nach und nach alle unter (5) deklarierten Typen innerhalb des Zielmodells instanziiert und die angegebenen Befehle ausgeführt. Im imperativen Teil können dabei Quellelemente mit ihren in (4) deklarierten Bezeichnern referenziert werden.

Jython-Blöcke

(6) zeigt noch eine Besonderheit der gATL: Zwischen den Schlüsselwörtern `JY:` und `YJ` können auf den deklarierten Quell- und Zielelementen beliebige *Jython-Befehle* ausgeführt werden. Damit ermöglicht es gATL bereits zu diesem frühen Zeitpunkt der

Implementierung, komplexe, turing-vollständige Transformationen zu beschreiben. Nach Meinung der Domain Engineers war die intuitive Konzeption der Python-Sprache ein geeigneter Kompromiss zwischen dem relativ hohen Aufwand der kompletten Realisierung der ATL zu diesem Zeitpunkt und dem Bedarf nach einer Transformationssprache, die in ihrer Ausdruckskraft den Ansprüchen eines Realprojekts genügt.

Referenz und Grammatik

Abschließend wollen wir der Vollständigkeit halber noch die Grammatik der gATL in EBNF angeben (Abb. 9.56). Da die Grundlagen der EBNF bereits im Transformationskapitel erklärt wurden und der Aufbau von gATL Transformationen mit dem vorigen Beispiel ausführlich beschrieben wird, wollen wir an dieser Stelle nicht näher auf Einzelheiten eingehen sondern sofort mit der Beschreibung der weiteren Aufgaben fortfahren, die noch zu lösen sind.

```
Identifier -> Letter ( Letter | Digit )*
Letter -> ["$", "A"-"Z", "_", "a"-"z"]
Digit -> ["0"-"9"]

START -> TransformationModule

TransformationModule ->
       ModuleIdentifier
       MMSpecification
       RuleList

ModuleIdentifier -> "module" Identifier
MMSpecification ->
       "create" "out" ":" Identifier "from" "in" ":" Iden-
tifier

RuleList -> ( Rule )*
Rule -> "rule" Identifier "{"
              "from" SourcePattern
              "to" TargetAction
         "}"

SourcePattern ->
       ( FullDeclaration [ "(" ( Expression )+ ")" ] )+
TargetAction ->
       ( FullDeclaration [ "(" ( Statement )+ ")" ] )+

FullDeclaration ->
       Identifier ":" Identifier "!" Identifier
Expression ->
```

Abb. 9.56
gATL Grammatik in EBNF

Abb. 9.56
gATL Grammatik
in EBNF (Forts.)

```
       ( Equality
               |
         "JY" ":" JythonBlock "YJ"
       ) ";"
Statement ->
       ( Assignment
               |
         "JY" ":" JythonBlock "YJ"
       ) ";"
Equality -> Identifier "=" Identifier
Assignment -> Identifier "<-" Identifier
```

Wir haben bereits angesprochen, dass die Arbeit des Transformators auf einer internen Objektbaumrepräsentation des in gATL verfassten Transformationsmoduls stattfindet. Die nächste Aufgabe wird also sein, an diesen Baum zu gelangen. Die Lösung des Problems erläutert der folgende Abschnitt.

9.5.3 Verarbeitung von gATL Transformationsmodulen

Wie bereits erwähnt, werden innerhalb des genua-Frameworks die Vorschriften, nach denen eine Transformation durchgeführt wird, in sogenannten *Transformationsmodulen* (synonym: Übersetzungseinheiten, ÜE) spezifiziert und dem Transformator in Form von Textdateien geliefert. Aus dieser Eingabe muss dann eine interne *Zwischenrepräsentation (engl. intermediate representation)* in Form eines Objektnetzes erzeugt werden, auf der dann die Algorithmen des Transformators arbeiten können. Um dies zu bewerkstelligen ist das technische Problem zu lösen, wie die vorerst in rein textueller Form vorliegenden Module für das Framework nutzbar gemacht werden können.

Das Problem kann in zwei Hauptbestandteile zerlegt werden:

- Die Datei muss in ihre sprachlichen Bestandteile zerlegt werden, was der *lexikalischen Analyse* der Transformationsbeschreibung entspricht.
- Es muss getestet werden, ob die ÜE den Produktionsregeln der gATL-Grammatik genügt. Dies entspricht einer *syntaktischen Überprüfung* bzw. dem *Parsen* der Datei.

Abb. 9.57 zeigt den Ablauf der Teilschritte noch einmal im Aktivitätsdiagramm. Die dargestellte Aktivität entspricht dem Verhalten,

das im Gesamttransformationsprozess in der Aktion `gATL - Processing` aufgerufen wird (vgl. Abb. 9.54).

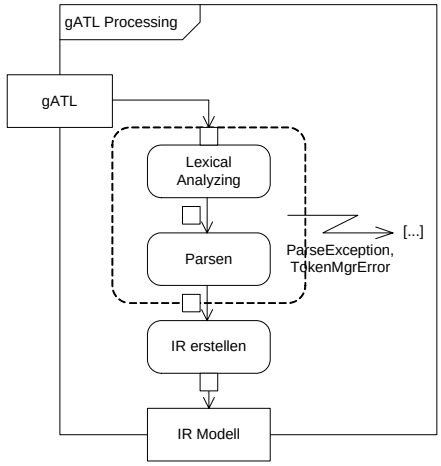


Abb. 9.57 Aktivität gATL Processing

Technische Realisation mittels JavaCC

Zum Glück müssen die im vorigen Abschnitt genannten Funktionsblöcke nicht auf der grünen Wiese realisiert werden. In genua wurde das Tool *JavaCC* [JavaCC] verwendet, das einen großen Teil der Aufgaben bequem und zuverlässig erledigen kann.

JavaCC ist das Java Pendant zu Yacc [Yacc]. Mit ihm kann zu einer gegebenen Grammatik der Java-Sourcecode des zugehörigen *Lexikalischen Scanners (Lexical Analyzer, LA)* sowie eines *Parsers* generiert werden. Aufgabe des LA ist die Zerlegung eines Eingabestromes in so genannte *Token*, die die zulässigen Wörter und Zeichen einer Sprache darstellen. Die so gefundenen Token dienen in Form einer geordneten Sequenz als Eingabe des *Parsers*, der dann an Hand der gegebenen *Grammatik* entscheidet, ob die Eingabe zu der zugehörigen *Sprache* gehört. Er führt damit die *syntaktische Überprüfung* durch.

Sowohl die Spezifikation der gültigen Syntax der Sprache, als auch die Angabe der Grammatik, geschehen bei JavaCC in ein und derselben Datei. Durch Aufruf von JavaCC werden die benötigten Java-Klassen generiert, die nach Fertigstellung dann in die eigene Applikation integriert oder direkt von der Kommandozeile ausgeführt werden können.

Definition der Grammatik

Erster Schritt zur Generierung des Analyzers und Parsers ist also die Spezifikation der Grammatik sowie die Definition der erlaubten Eingabewörter (Token). Die Beschreibung geschieht dabei in einer JavaCC-spezifischen Mischung aus EBNF-ähnlichen Konstrukten und Java-Code. Am besten machen wir uns das Ganze an einem Beispiel deutlich:

9.5.4 Beispiel

Vor der Arbeit mit JavaCC steht die Installation des Tools. Diese geschieht im Falle von JavaCC denkbar einfach. JavaCC liegt momentan in Version 3.2 vor und kann unter [JavaCC] beispielsweise als `javacc-3.2.zip` heruntergeladen werden.

Nach dem Entpacken in das gewünschte Zielverzeichnis muss lediglich das `/bin` Verzeichnis der Distribution in den Systempfad aufgenommen werden. Danach ist das Programm durch `javacc [optionen]` aufrufbar.

Als Basis des Beispiels dient uns die in
Abb. 9.58 dargestellte Grammatik. Sie liegt im EBNF-Format vor und soll nun in eine von JavaCC auswertbare Form umgesetzt werden.

Abb. 9.58 Beispielgrammatik

```
Name    := Letter (Letter | Digit)*
Letter  := "A"-"Z" | "a"-"z"
Digit   := "0"-"9"
Hello   := "I" "am" Name "."

<Start> := Hello
```

Die Aufgabe

Resultat der Bemühungen soll ein Programm sein, das für uns die Begrüßung von Neuankömmlingen in der MDA-Gemeinde übernimmt. Begrüßt werden soll jeder, der uns in anständiger Form seinen Namen mitteilt. Die obige Grammatik verrät uns in diesem Zusammenhang, was dabei unter „anständig“ verstanden werden soll. Erlaubt sind Sätze wie „*I am Herbert.*“ aber auch „*I am C3PO.*“. Aufgabe des Programms ist es nun zu entscheiden, ob unser Gegenüber sich korrekt vorgestellt hat und, wenn dem so sein sollte, den Namen herauszubekommen, damit es zurückgrüßen kann.

Abb. 9.59 zeigt die JavaCC-Spezifikationsdatei, aus der später die Generierung der Hilfsklassen geschehen soll, die dann die Erken-

nung einer anständigen Vorstellung, die Extraktion des Namens und letztendlich auch die Begrüßung übernimmt.

Dabei soll vor allem das Grundprinzip der Verwendung von JavaCC verdeutlicht werden: Produktionsregeln, die ursprünglich in EBNF vorliegen, werden zur Verwendung innerhalb JavaCC in eine hybride Java-/EBNF-Syntax gebracht. Die Beschreibung erinnert in ihrer Form gewöhnlichen Methoden, die von JavaCC genau dann aufgerufen werden, wenn die entsprechenden Token(-Kombinationen) im Eingabestrom durch den Parser entdeckt werden. Gleichzeitig kann Java-Code spezifiziert werden, der bei Aufruf der bei Aufruf der jeweiligen Methode ausgeführt werden soll. Hier werden später genau die Code-Teile stehen, die die zum gefundenen Token gehörenden Objekte des Zwischenmodells instanziieren.

Das Beispiel dient vor allem als erste Einführung in die Thematik. Eine genauere Beschreibung der Funktionsweise der einzelnen Bausteine erfolgt dann im weiteren Verlauf des Kapitels.

Abb. 9.59
JavaCC Beispiel

```
/*
  myGrammar.jj
*/
options {
    IGNORE_CASE=true;
}

PARSER_BEGIN(HelloParser)
    import java.io.*;

    public class HelloParser {                          (1)
        public static void parse(InputStream in)
        throws ParseException, TokenMgrError {

            String name = null;
            HelloParser parser = new HelloParser(in);
            name = parser.Hello();                      (2)
          System.out.println("Hello, "+name);
        }
    }

PARSER_END(HelloParser)

SKIP:  {" "}
SKIP:  {"\n" | "\r" | "\r\n"}

// production rules
String Hello():                                         (3)
{
   Token nameToken = null;
```

Abb. 9.59 JavaCC Beispiel (Forts.)

```
}
{
   "I" "am" nameToken = <NAME> "."                    (4)
   {return nameToken.image;}                          (5)
}

// token definitions
TOKEN: {
    <NAME: <LETTER>( <LETTER> | <DIGIT> )*>
    | <#LETTER: ["A"-"Z","a"-"z"]>
    | <#DIGIT: ["0"-"9"]>
}
```

Den Einstiegspunkt der Anwendung bildet die Methode, die mit der als Anfangsregel gekennzeichneten Produktionsregel korrespondiert (3). Sie wird innerhalb der generierten Parser-Klasse (hier `HelloParser`) (1) manuell aufgerufen (2). Der Aufruf der restlichen Methoden geschieht dann entsprechend der gefundenen Token rekursiv von diesem Punkt aus. Wurde eine komplette Vorstellung erkannt, liefert die Methode den Namen zurück, die in der Vorstellung als `<NAME>`-Token erkannt wurde (4+5).

Abb. 9.60 Generierung der Hilfsklassen

```
Java Compiler Compiler Version 3.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file myGrammar.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist. Will create one.
Parser generated successfully.
```

Der Schritt, der der Erstellung der Grammatik folgt, ist die Generierung von Hilfsklassen durch JavaCC auf Basis der zuvor erstellten Definitionen. Der Aufruf von `javacc myGrammar.jj` in der Kommandozeile liefert den in Abb. 9.60 dargestellten Output. Die fettgedruckten Dateien werden von JavaCC generiert und bilden das von uns erwartete Ergebnis. Nicht in der Ausgabe erwähnt, jedoch ebenso erstellt werden folgende Klassen:

- `HelloParser.java`
- `HelloParserConstants.java`
- `HelloParserTokenManager.java`

Besonders interessant für uns: die Klasse `HelloParser`, die die von uns im Spezifikationsfile (Abb. 9.59) angegebene Methode

`parse(…)` (1) enthält, und die Schnittstelle zwischen den JavaCC-generierten Artefakten und unserer Anwendung darstellt.

```
/*
  JavaCCTest.java
*/

public class JavaCCTest {

   public static void main(String[] argv) {
      try {
         String name =                                    (1)
         HelloParser.parse(System.in);
         System.out.println ("Hello, " + name + " !"); (2)
      }catch (Exception e) {                           (3)
         e.printStackTrace();
     }
    }
}
```

Abb. 9.61 Einbinden des generierten Parsers

Abb. 9.61 zeigt, wie wir die generierten Klasse in unserer Applikation nutzen können. Nach erfolgreichem Durchkompilieren und Starten des Programmes kann der Parser durch Eingabe eines Strings ausprobiert werden. Dabei sollte herauskommen, dass die Methode `HelloParser.parse(InputStream)` (1) alle Eingabeströme akzeptiert, die unserer Grammatik entsprechen und bei korrekter Syntax den Namen desjenigen zurückliefert, der begrüßt werden soll. Die Begrüßung selbst findet in (2) statt. Bei fehlerhaften Eingaben wird eine entsprechende `Exception` geworfen (3), auf die dann mit einer geeigneten Fehlermeldung reagiert werden kann.

Voilà …

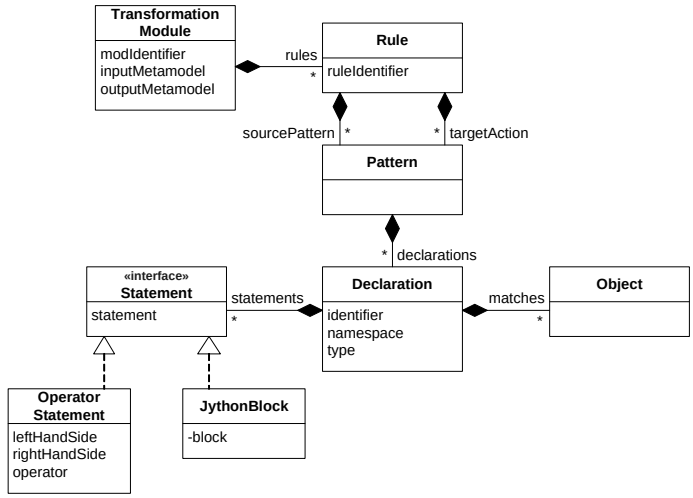
Schauen wir kurz auf das eben vollbrachte zurück, stellen wir fest, dass wir soeben den ersten Teil unserer momentanen Aufgabe gelöst haben: die Überprüfung auf syntaktische und grammatikalische Fehler der Transformationsbeschreibungen.

9.5.5 Konstruktion des Zwischenmodells

Widmen wir uns nun dem weiteren Verlauf des Wegs zur Erstellung des Zwischenmodells und betrachten zuerst die statische Struktur des zu erzeugenden Modells (Abb. 9.62). Beim Vergleich des Diagramms mit der in Abb. 9.56 gegebenen Grammatik fällt bei näherer Betrachtung die Ähnlichkeit des Aufbaus mit den dort formulierten Produktionsregeln auf. Sie sind nahezu identisch, wobei lediglich einige Vereinfachungen zur Verschlankung des Designs vorgenom-

men wurden. Hinzugekommen ist beispielweise die Komposition einer Liste von *Treffern* innerhalb einer *Deklaration*. Hier werden an späterer Stelle der Transformation die gefundenen Quellelemente gespeichert und können von dort vom Transformator zur weiteren Verarbeitung bezogen werden.

Abb. 9.62 gATL Zwischenmodell



Nachdem die Struktur des verwendeten Zwischenmodells klar ist, kommt die ursprüngliche Frage wieder in den Sinn: Wie baut man denn nun das Objektnetz konkret aus der textuellen Repräsentation zusammen? Antwort: durch die Verwendung von JavaCC. Schauen wir uns dazu die folgende Abbildung an, die einen Ausschnitt aus der aktuellen JavaCC-Grammatik der gATL darstellt (Abb. 9.63).

Abb. 9.63 Konstruktion des IR-Objektbaums mittels JavaCC

```
PARSER_BEGIN(GATL)

package genua.framework.transformator.parser;

import genua.framework.transformator.model.*;
import java.io.InputStream;
                                                          (1)
public class GATL {
 public static TransformationModule
      parseModule(InputStream in)
      throws ParseException, TokenMgrError {
         GATL parser = new GATL(in);
         return parser.TransformationModule();            (2)
 }
}

PARSER_END(GATL)
```

```
[…]

/* gATL grammar */

TransformationModule TransformationModule():                    (3)
{
    String moduleID;
}
{
    moduleID = ModuleIdentifier()                               (4)
        {
            TransformationModule module =
                      new TransformationModule(moduleID);
        }
    MMSpecification(module)                                     (5)
    RuleList(module)                                            (6)

    { return module; }
}[…]
void RuleList(TransformationModule module):                     (7)
{
    Rule rule = null;
}{
    (
        rule = Rule()                                           (8)
            {
                module.getRules().add(rule);                    (9)
            }
    )*                                                         (10)
}
Rule Rule():
{
    Rule rule = new Rule();
    Token ruleID = null;
    Pattern source, target = null;
}
{
    "rule" ruleID = <IDENTIFIER> "{"
    "from" source = SourcePattern()                            (11)
    "to"   target = TargetAction()
    "}"
    {
        rule.setRuleIdentifier(ruleID.image);                  (12)
        rule.setSourcePattern(source);
        rule.setTargetAction(target);
        return rule;
    }
}
```

Abb. 9.63 Konstruktion des IR-Objektbaums mittels JavaCC (Forts.)

In (1) findet sich der Bereich der Definition, der später in Form einer Java-Klasse die Schnittstelle zu unserer Applikation (dem Transformator) darstellen wird. Diese Klasse heißt `GATL`, die statische Methode, die das IR-Modell zurückliefert und gleichzeitig die Überprüfung auf Korrektheit erledigt, kann unter dem Namen `GATL.parseModule(…)` aufgerufen werden. Da die Klassen des IR-Modells im Package `genua.framework.transformator.model` untergebracht sind, muss dieses in die Import-Liste aufgenommen werden.

(2) zeigt, dass der letztendlich an unsere Anwendung zurückgelieferte Wert des Rückgabewert der Methode `TransformationModule()` entspricht und in einer Instanz der Klasse gleichen Namens resultiert, die dem Wurzelobjekt des IR-Modells entspricht. In der zurvor betrachteten EBNF (Abb. 9.56) hieß die Startproduktionsregel ebenfalls `TransformationModule`, eine Gleichheit, die ein erstes Muster der Umsetzung von EBNF in JavaCC deutlich werden lässt.

In (3) findet sich die „Methode“ `TransformationModule()`. Sie stellt den Einstiegspunkt des Parsing-Vorgangs dar, und ruft ihrerseits die Methoden `ModuleIdentifier()`, `MMSpecification()` und `RuleList()` auf (4+5+6). Man vergleiche auch hier mit der EBNF, in der folgende Ableitungsregel existierte:

```
TransformationModule ->
       ModuleIdentifier
       MMSpecification
       RuleList
```

Das Vorgehen von Abbildungen von EBNF Produktionsregeln in JavaCC besteht also darin die Regeln als Methoden aufzufassen, die bei Auftreten der passenden Token im Eingabestrom aufgerufen werden. Der rekursive Charakter der EBNF wird durch die passende Verschachtelung von Methoden abgebildet. Ausgehend von der Startmethode werden solange weitere Methoden aufgerufen, bis ein gültiger Endzustand erreicht ist oder eine Verletzung der Produktionsregeln vorliegt, was dann in einer `Exception` resultiert.

Obwohl wir an dieser Stelle kein vollständiges Tutorial für die Verwendung von JavaCC liefern können und dies der angegebenen Literatur vorbehalten müssen, wollen wir noch einige Erläuterungen zu den verwendeten Konstrukten liefern, um das Beispiel zumindest intuitiv verständlich zu machen:

Rückgabewerte können in lokalen Variablen, die im ersten Block einer Methode/Regel deklariert wurden, gespeichert werden. Wie erwähnt werden Methoden genau dann aufgerufen, wenn die entsprechenden Token im Eingabestrom auftauchen. So wird bei erkanntem Beginn einer Regel die entsprechende Methode `Rule()` aufgerufen (8), die ihrerseits bei Beginn der Quellelement-Deklaration die Methode `SourcePattern()` aufruft (11), usw. Kehrt die Kontrolle nach beendeter Rekursion zurück, werden zusätzlich die Anweisungen im, der Methode folgenden Block ausgeführt, so z. B. das Setzen des Regelnamens im Zwischenmodell in Zeile (12), Abb. 9.63.

(7)+(8)+(9) zeigen so den Zusammenbau der Liste von Regeln. Die Methode/Produktionsregel `Rulelist()` gibt dabei nicht eine Liste von Regeln zurück, sondern bekommt das aktuelle `TransformationModule` als Parameter per Referenz übergeben und fügt, wie wir gleich sehen werden, die gefundenen Regeln direkt dieser Datenstruktur hinzu (9).

(10) verdeutlicht noch einmal die Vermengung von Java- und EBNF-Syntax. Innerhalb einer Übersetzungseinheit können beliebig viele Regeln angegeben werden. Dies wird durch die Klammerung des Aufrufs mit folgendem „`*`“ dargestellt. Die Anweisung `rule = Rule()` (8) wird also für jede gefundene Regel durchgeführt und diese nach jedem einzelnen Aufruf dem `TransformationModule` hinzugefügt (9).

Weiterführende Literatur

Auf der JavaCC Homepage [JavaCC] finden sich sowohl die Realisierung einer Fülle weiterer Beispielgrammatiken als auch Tutorials und Referenzen, die alle verfügbaren Features erklären und das Selbststudium ermöglichen. Unter [Nor04a] bzw. [Nor04b] findet man zum einen das einführende Kapitel eines JavaCC Tutorials und zum anderen die JavaCC FAQs, die viele Fragen, die dem JavaCC-Neuling das Leben schwer machen können, ausführlich beantwortet.

9.5.6 Deserialisierung des Quellmodells

Als letzter Schritt vor der eigentlichen Transformation bleibt noch die *Deserialisierung* (sprich das „Laden“) des Quellmodells zu erledigen. Diese wird für uns jedoch vollständig vom benutzten UML2-Framework des Eclipse Tool-Projects übernommen, das wir bereits im Technologie-Plan der Projektplanung (Kapitel 8) beschrieben haben. Deshalb gehen wir im Folgenden davon aus, dass die Zwi-

schenrepräsentation des Quellmodells bereits als UML2-/EMF-Objektnetz vorliegt, und fahren mit der Beschreibung des Transformationsvorgangs fort.

9.5.7 Durchführung der Transformation mittels Jython

Nachdem wir die technischen Herausforderungen der vorangegangenen Abschnitte gemeistert haben, können wir uns nun an das Kernstück der Transformation wagen.

Was ist zu tun? Folgende Schritte stehen noch an:

- Das *Auffinden aller Elemente*, die die in gATL formulierten Bedingungen erfüllen und
- die *Erstellung des Zielmodells* durch Instanziierung der ebenfalls dort deklarierten Zielelemente und die Ausführung der spezifizierten Zielaktionen auf dem Zielmodell.

Die durchzuführenden Zielaktionen werden *target actions* genannt. Da man die Bedingungen an die Quellelemente auch als eine Art Muster betrachten kann, dem die zur Verfügung stehenden Kandidaten entsprechen müssen, wird im Folgenden die Aufgabe der Kandidatenermittlung als *Mustererkennung (engl. pattern matching)* bezeichnet.

Beide Aufgaben haben gemein, dass ihre Realisierung im Falle des genua Frameworks wesentlich auf der Verwendung von *Jython* als unterliegender Technologie basiert. Sowohl die Bedingungen an mögliche Quellelemente als auch die auf den tatsächlich gefundenen Quellelementen durchgeführten Aktionen, die in den Transformationsregeln im gATL-Format spezifiziert wurden (vgl. Abschnitt 9.5.2), werden zuerst auf Jython-Instruktionen abgebildet und dann erst als solche ausgeführt bzw. die Ausdrücke zu `true` bzw. `false` ausgewertet.

Beginnen wir damit uns die Grundlagen von Jython ein wenig näher anzuschauen.

Jython Basics

Jython [Jython] stellt eine Implementierung der Skriptsprache Python [Python] in Java dar. Innerhalb von Java-Applikationen können Jython-Skripte ausgeführt und Objekte beider Welten über die Nutzung einer genau definierten Schnittstelle gemeinsam verwendet

werden. Durch diese nahtlose Integration beider Sprachen können die Vorteile beider Technologien voll ausgenutzt werden:

- Die weite Verbreitung und die mächtige Sammlung von Anwendungsbibliotheken (APIs) von Java und
- die Flexibilität und „Einfachheit“ von Python.

Obwohl Python eine vollwertige Programmiersprache ist, sind die dort verwendeten Sprachelemente für jeden Programmierer intuitiv einleuchtend und somit mit niedriger Lernkurve sofort anwendbar. Vorhandene Java-Klassen können in Jython-Skripte importiert und die in ihnen enthaltenen Attribute im Jython-Kontext verwendet werden. Auch ein Aufruf der Methoden der so übergebenen Objekte aus Jython-Skripten ist möglich, was eine vollständige Integration ermöglicht. Die angesprochene Flexibilität Jythons entspringt vor allem dem dynamischen Charakter dieser Sprache. Da keine expliziten Typendeklarationen notwendig sind, sondern Jython die übergebenen Java Objekte zur Laufzeit mit Hilfe seiner introspektiven Fähigkeiten analysiert, kann einfach und unkompliziert auf die so gefundenen Attribute und Methoden zugegriffen werden.

Bei der Verwendung im Kontext der Transformation kommt vor allem diese Fähigkeit Jythons zum Tragen, Eigenschaften der importierten Java Klassen dynamisch durch Introspektion zu ermitteln und dann die angegebenen Instruktionen auszuführen beziehungsweise die in Form boolescher Ausdrücke an die Elemente gestellten Anforderungen auszuwerten.

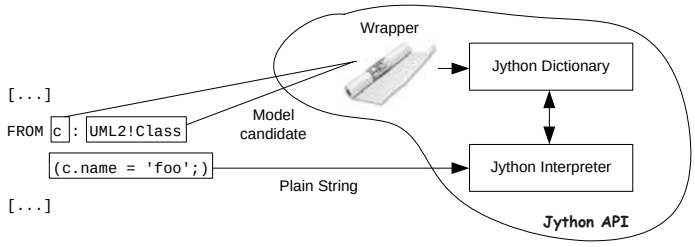


Abb. 9.64 Ausführung von GTL Anweisungen in Jython

Abb. 9.64 verdeutlicht den prinzipielle Einsatz von Jython im Framework: Ausgehend von der Deklaration der `FROM`-Klausel (hier: `c:UML2!Class`), werden innerhalb einer Java-Methode zuerst die möglichen Kandidaten des Quellmodells ermittelt, was in diesem Beispiel allen Modellelementen vom Typ `Class` im Namensraum `UML2` entspricht. Für jeden einzelnen der Kandidaten werden dann die Bedingungen überprüft, die für ein positives Matching erfüllt sein müssen. Im vorliegenden Beispiel also alle Elemente die den

Namen `foo` besitzen. Diese Überprüfung findet nun mit Hilfe des Jython-Interpreters statt.

Dictionary Datenstruktur

Da nach Verwendung von EMF/UML2 in genua das Quellmodell in Form eines Objektbaums vorliegt, bestehen die gefundenen Kandidaten aus Objekten dieses Baumes. Diese werden in spezielle Jython- Klassen gewrappt, und dann im *Jython Dictionary* eingetragen. Das Dictionary entspricht dem *Variablenraum* des Jython Interpreters und stellt die Schnittstelle zwischen Python- und Java-Welt dar. Objekte der Java-Welt können dort eingetragen, und dann vom Interpreter als Python-Objekte verwendet werden. Durch den unterliegenden Sandbox-Charakter wird so eine saubere Trennung der beiden Technologien erreicht. Sollten während der Ausführung der Python-Skripte Veränderungen an den übergebenen Objekten stattgefunden haben, kann auch die Java Anwendung am Ende der Skriptausführung auf diese durch das Dictionary zugreifen und so die Änderungen wieder in die Java-Welt „herüberretten".

Technisch gesehen besteht das Dictionary aus einer Jython-eigenen Hashmap-Implementierung, in der der Variablennamen den *Schlüssel* darstellt und das übergebene Objekt dem *Wert* entspricht. Der Interpreter-Klasse wird bei Instanziierung das erzeugte Dictionary-Objekt im Konstruktor übergeben und letztlich die Funktion zur Ausführung der in Jython formulierten Befehle bzw. der Auswertung von booleschen Ausdrücken auf der Interpreter-Instanz aufgerufen.

In unserem Beispiel in Pseudocode (Abb. 9.65) geschieht die Übergabe eines Einzelausdrucks im `String` Format an die `eval(String)` Methode des Interpreters. Der Interpreter wertet den Ausdruck anhand der im Dictionary verfügbaren Daten aus, und übergibt den ermittelten Wert, wiederum als `String`, an den Aufrufenden zurück, der das Ergebnis dann weiter verarbeiten kann.

Ermittlung der Quellelemente

Bei der Ermittlung der Elemente, die alle Bedingungen erfüllen und so an der Transformation teilnehmen (Quellelemente) bedeutet dies konkret (Abb. 9.65):

Abb. 9.65 Algorithmus zur Ermittlung der Quell-Elemente

```
Für alle Kandidaten:
       Trage Kandidat in Dictionary ein
       Für alle Bedingungen:
              Rufe interpreter.eval(Bedingung) auf
              Ist Rückgabewert WAHR: Behalte Kandidaten
              Sonst: Entferne Kandidat
                     Weiter mit nächstem Kandidaten
```

Wie besprochen besteht die Menge der Kandidaten dabei initial aus allen Elementen des Quellmodells, die dem geforderten Typ im angegebenen Namensraum der `FROM`-Klausel entsprechen. Das Quellmodell liegt zu diesem Zeitpunkt bereits als (IR-)Objektbaum vor. Nach und nach werden alle Kandidaten für die mindestens eine Bedingung nicht erfüllt ist aus der Menge entfernt, sodass am Ende nur die tatsächlichen Quellelemente übrig bleiben.

Die so gefundenen Elemente werden zusammen mit den zugehörigen Deklarationen im IR-Modell gespeichert und an die zweite Stufe des Transformators weitergereicht, der dann die Zielelemente erzeugt, und die Zielaktionen auf ihnen durchführt.

Durchführen der Zielaktionen (target actions)

Die folgende Abbildung zeigt die notwendige Vorgehensweise für eine einzelne Regel (Abb. 9.66).

```
Für alle Quelldeklarationen
        Für alle Quellelemente
                Für alle Zielaktionen
                        Erzeuge Zielelement
                        Trage Hilfsklassen in Dictionary ein
                        Trage Zielelement in Dictionary ein
        Trage Quellelement in Dictionary ein
        Für alle Zielstatements
                Erzeuge Jython-Syntax
                Konkateniere Statements
                Rufe interpreter.exec(statementblock) auf
```

Abb. 9.66 Algorithmus zur Durchführung der Zielaktionen

Für alle pro Quellmuster (-deklaration) gefundenen Quellelemente werden alle entsprechenden Zielaktionen durchgeführt. Dazu wird das in der Zielaktion deklarierte Zielelement im Objektbaum des Zielmodells erzeugt und zusammen mit dem korrespondierenden Quellelement sowie einiger Hilfsklassen in das Dictionary des Jython-Interpreters eingetragen. Aus der Spezifikation der Zielaktionen wird ein entsprechender Jython-Codeblock zusammengebaut und dieser dann im Interpreter ausgeführt. Basis zum Zusammenbau der Codeblöcke bildet der Inhalt der gefundenen `Statements` (vgl. Abb. 9.62). In dieser ersten Iteration ist dies vor allem der Inhalt von Jython-Blöcken, die reinem Jython-Code entsprechen (vgl. Abb. 9.55 (6)). Nachdem alle Zielaktionen durchgeführt sind, ist der Transformationsvorgang abgeschlossen, und das Zielmodell steht zur weiteren Bearbeitung bzw. zum Speichern zur Verfügung.

9.5.8 Durchführung einer Transformation mittels Ant

Das Starten einer Transformation geschieht in genua durch die Ausführung eines eigenen Ant- [Ant] Tasks. Abb. 9.67 zeigt einen Ausschnitt aus einer `build`-Datei, die den Task namens `genua_Transformation` verwendet.

Abb. 9.67 Anwendung des Ant-Tasks

```
<target name="genua_transformation" depends="init">    (1)
   <taskdef name="genua_transformation"                 (2)
     classname="genua.framework.transformator.
                  ant.TransformationTask"
     classpath="${build.dir}"
   />
   <genua_transformation
      transformation_module =
       "${module.dir}/${transformation_module.filename}"
      source_model =
         "${model.dir}/${sourcemodel.filename}"
      target_model =
          "${model.dir}/${targetmodel.filename}"        (3)
      profiles_dir = "${profiles.dir}"
    />
</target>
```

Zuerst wird der Name des Tasks definiert und die Java-Klasse angegeben, die die Aufgabe der Transformation übernehmen soll.

Abb. 9.68 Parameter des Transformations-Tasks

Parameter	Beschreibung
transformation_module	Enthält Pfad und Namen der gATL-Datei, mit den Transformationsregeln, die bei der Transformation angewendet werden sollen.
source_model	Enthält Pfad und Namen der Datei, die das Quellmodell enthält.
target_model	Enthält Pfad und Namen der Datei, in der das transformierte Modell gespeichert werden soll.
profiles_dir	Enthält den Pfad zu dem Verzeichnis, in dem die zu verwendenden Profile gespeichert sind.

Dies geschieht in (1), indem der Task `genua_transformation` eingeführt und die Klasse `TransformatorTask` als verantwortlich für die Ausführung angegeben wird (2). Danach geschieht der eigentliche Aufruf des Tasks, der die Transformation auslöst (3). Die in Abb. 9.68 dargestellten Parameter müssen dabei übergeben werden.

Der Task führt die in Abschnitt 9.5 beschriebenen Schritte durch, die wir an dieser Stelle nicht noch einmal wiederholen wollen. Näheres zu Ant findet sich auf der Jakarta-Webseite unter [Ant].

Code-Snippets

Obwohl wir den Code des Transformators aus Platzgründen nicht vollständig angeben können, sollen im Folgenden einige Code-Snippets vorgestellt und kurz in loser Form erläutert werden, die während der Entwicklung des genua Transformators eine Rolle gespielt haben und einen Eindruck von der Verwendung von Jython liefern.

```
// Notwendige Importe                                    (0)
import org.python.core.Py;
import org.python.core.PyStringMap;
import org.python.util.PythonInterpreter;

// Initialisierung des Interpreters                      (1)
PythonInterpreter.
     initialize(preProperties, postProperties, argv);

// Dictionary erzeugen und mit Interpreter verbinden     (2)
PyStringMap dict      = new PyStringMap();
PythonInterpreter pip = new PythonInterpreter(dict);

// Variablen in's Dictionary eintragen                   (3)
pip.set(varName, Py.java2py(myJavaObject));

// Ausführen von Jython-Code                             (4)
pip.exec(myStatementString); // als String
pip.execfile(myInputStream); // als FileStream

// Auswerten (true/false) von Jython-Statements          (5)
PyObject myPO = pip.eval(myStatementString);
```

Abb. 9.69 Jython-Snippets

(0) Zu den notwendigen Imports muss wohl nicht mehr gesagt werden, als dass sie alle in `jython.jar` zu finden sind, in dem sich die gesamte Basis-API von Jython befindet.

(1) Bevor der Interpreter innerhalb der Java-Applikation genutzt werden kann, muss er initialisiert werden. Parameter der Initialisierung sind:

- `preProperties`: Werden im Normalfall mit `System.getProperties()` belegt.
- `postProperties`: Überlädt `preProperties`. Sowohl `preProperties`, als auch `postProperties` sind vom Typ `Property`, und können die in [JythonReg] angegebenen Werte annehmen.
- `argv`: Hier können etwaige Argumente aus der Kommandozeile übergeben werden. Im Normalfall ist hier die Übergabe von `new String[]` ausreichend.

(2) Das *Dictionary* repräsentiert die Schnittstelle, die es der Java-Applikation ermöglicht, Objekte in den Jython-Namensraum zu übergeben und so den Jython-Operationen zugänglich zu machen. Auch die Kommunikation in umgekehrter Richtung, beispielweise die Übernahme des erzeugten Zielmodells in die Java-Welt, findet über dieses Konstrukt statt.

(3) Der Eintrag eines Java-Objektes in das Dictionary entspricht dem Eintrag in eine *Hashmap*. Der *Schlüssel* (in Form eines `Strings`) repräsentiert den Variablennamen, unter dem das Objekt im Namensraum des Interpreters angesprochen werden soll. Bevor beliebige Objekte eingetragen werden können, müssen diese in eine spezielle Wrapperklasse gehüllt werden. Um dies zu bewerkstelligen, bietet Jython die Funktion `Py.java2Py()`, die genau diese Aufgabe übernimmt.

(4) Die vom Interpreter auszuführenden Befehle, können entweder als `String` oder in Form eines *Streams*, der ein *Python-Sourcefile* zugänglich macht, übergeben werden. Sind Objekte, die vor der Ausführung mittels des Dictionary an den Interpreter übergeben wurden, durch die ausgeführten Befehle geändert worden, steht ihr neuer Zustand der umschließenden Java-Methode ganz normal über die ursprüngliche Referenz (in diesem Beispiel `varName`) oder alternativ über die `get(key)`-Methode des Dictionaries zur Verfügung.

(5) Soll ein einzeiliger Ausdruck ausgewertet werden, kann die `eval()`-Operation des Interpreters verwendet werden. Das Ergebnis wird in Form eines `PyObject` zurückgeliefert. In ihrer rudimen-

tärsten Form können `PyObjects` dabei als Strings ausgewertet werden. Bei der Auswertung boolescher Ausdrücke bedeutet ein Wert von „`1`“ `true` bzw. „`0`“ `false`.

Bezugsquellen und Einstieg

Die neueste Version von Jython – zur Zeit der Drucklegung ist dies die Version 2.1 – kann auf der Jython Homepage heruntergeladen werden [Jython]. Jython wird in Form einer `.class` Datei ausgeliefert, die zur Installation lediglich auf dem Rechner ausgeführt werden muss (Beispiel: `java jython-21`). Das weitere Vorgehen geschieht dann assistentengeführt, mögliche Probleme und deren Lösungen sind ausführlich auf der Homepage beschrieben.

Ebenfalls auf der Homepage verlinkt sind eine Fülle von Tutorials, Beispielen und FAQ-Sammlungen, die es ermöglichen tiefer in die Welt der Jython-Programmierung einzusteigen. Die JavaDocs der aktuellen Version sind unter [JythonAPI] zu finden. Viele Details und weiterführende Links zur Sprache Python selbst finden sich unter [Python].

9.6 genua Model2Text-Transformator (gM2T)

Die Generierung textueller Artefakte ist in genua, wie auch die zuvor beschriebene Modell-zu-Modell Transformation, mittels eines parametrisierten Ant-Tasks realisiert. Abb. 9.70 zeigt einen Ausschnitt aus einer entsprechenden `build.xml`.

```
[…]
<target name="genua_generation" depends="init">          (1)
    <taskdef name="genua_generation"
        classname="ant.GenerationTask"
        classpath="${build.dir}"
    />
    <genua_generation                                    (2)
        source_model   = "…"
        config_dir     = "…"
        generator_base = "…"
        generators     = "JavaClassCodeGenerator"
        target_model   = "…"
    />
</target>
[…]
```

Abb. 9.70 gM2T – Generation Task

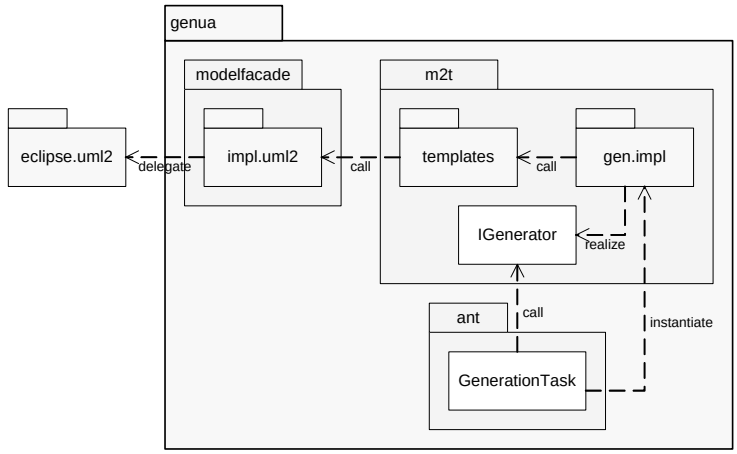
In (1) ist der Name des Tasks „`genua generation`“ sowie die definierende Klasse `ant.GenerationTask` erkennbar, die die

Logik des Tasks enthält. Beim Aufruf können folgende Parameter übergeben werden (2):

- `source_model`: Der Pfad des Quellmodells, das als Basis der Generierung dient.
- `config_dir` : Angabe des Verzeichnisses, in dem (optional) Konfigurationsdateien für die einzelnen Generatoren abgelegt werden können.
- `generator_base` : Das Wurzelverzeichnis, in dem die generierten Textdateien abgelegt werden.
- `generators` : Eine kommaseparierte Liste der auszuführenden Generatoren. Die hier angegebenen Namen entsprechen den (un)qualifizierten Klassennamen der Generationsklassen.
- `target_model`: Pfad und Name des zu erstellenden Zielmodells

Betrachten wir nun schematisch die Architektur des Generators, um die genaue Funktionsweise näher zu erörtern (Abb. 9.71):

Abb. 9.71
gM2T Architektur



Unten in der Abbildung findet sich die bereits erwähnte Klasse `GenerationTask`, die innerhalb gM2T Ant-Tasks aufgerufen wird und die beschriebenen Parameter übergeben bekommt. Prinzipiell funktioniert die Generierung der Text-Artefakte nun so, dass die in `generators` spezifizierten Generatorklassen nacheinander reflektiv instanziiert und dann über eine spezielle Methode des, allen Generatoren mandatorischen, Interfaces `IGenerator` aufgerufen wird, die die Erstellung der Artefakte übernimmt. Betrachtet man

unter diesem Gesichtspunkt die o. a. Architektur genauer, wird die Bedeutung des Packages `m2t.gen.impl` deutlich: hier befinden sich die Implementierungen der einzelnen Generatoren. Dem Beispiel der vorigen Abbildung (Abb. 9.71) nach wäre hier also eine Klasse `JavaClassCodeGenerator` zu finden.

Die Generatoren arbeiten auf dem übergebenen Quellmodell in EMF-/UML2-Form und erzeugen im nächsten Schritt die textuellen Artefakte. Der dazu verwendete template-basierte Ansatz und die dort benutzten Technologien werden im Folgenden näher erläutert.

9.6.1 Java Emitter Templates (JET)

JET ist Teil des Eclipse Modeling Frameworks (EMF) [EMF] und realisiert in genua die template-basierte Erzeugung textueller Artefakte. Abb. 9.72 zeigt ein `HelloWorld`, das die grundsätzliche Vorgehensweise bei der Benutzung von JET innerhalb des Frameworks verdeutlicht:

```
<%@ jet                        (1)
    package="hello"
    class  ="HelloWorldTemplate"
%>

Hello <%=argument%> !
```

helloworld.txtjet

```
[...]
                                                    (2)
HelloWorldTemplate hwt = new HelloWorldTemplate();
String result = hwt.generate("World");
System.out.println(result);                         (3)

[...]
```

java

```
Hello World ! (4)
```

output

Abb. 9.72 JET Hello World Anwendung

Die ersten Zeilen des Beispiels (1) zeigen bereits das Grundprinzip der JET-Technologie: JET-Templates werden vor der Benutzung in Java-Klassen übersetzt und können dann aus jeder Java-Applikation aufgerufen werden, um die Generierung der textuellen Artefakte anzustoßen. Der Parameter `package` zeigt an, welchem Package die generierte Klasse angehören soll, `class` spezifiziert den Namen der generierten Klasse (hier: `HelloWorldTemplate`). Bei Einsatz von Eclipse als IDE geschieht die Übersetzung der Templates beim

Abspeichern der Template-Datei automatisch. Die Übersetzung kann mittels JET-API jedoch auch manuell angestoßen werden.

Nach erfolgreicher Übersetzung kann dann an beliebiger Stelle einer Java-Anwendung die generierte Template-Klasse aufgerufen und so der Text generiert werden. (2) zeigt die Instanziierung und den Aufruf der `generate` Methode. Zur Parametrisierung der Templates erlaubt JET die Übergabe eines Parameters vom Typ `Object`. Im Beispiel ist dies der String `World`. Abschließend wird in (3) der Output des JET-Aufrufs – in Form eines Strings – übernommen und ausgegeben: `Hello World!`

Bevor wir mit der Erläuterung der Code-Generierung in genua fortfahren, wollen wir erst ein weiteres Konzept einführen und erläutern, das bei diesem Vorgang Verwendung findet – die Verwendung von *Modell-Fassaden*.

9.6.2 Modell-Fassaden

Auch wenn es noch nicht explizit angesprochen wurde ist intuitiv klar, dass die Code-Erzeugung, wie bereits bei Transformationen ausführlich betrachtet, mittels eines Quellmodells stattfinden muss, das den JET-Templates (siehe voriger Abschnitt) übergeben wird und als Basis zum Füllen der „Lücken“ innerhalb der Templates dient. Um von dem Typ eines konkreten Zwischenmodells zu abstrahieren und die generelle Lesbarkeit der Templates zu verbessern wurde der Einsatz einer Modell-Fassade beschlossen, die genau diese Aufgaben erfüllen soll.

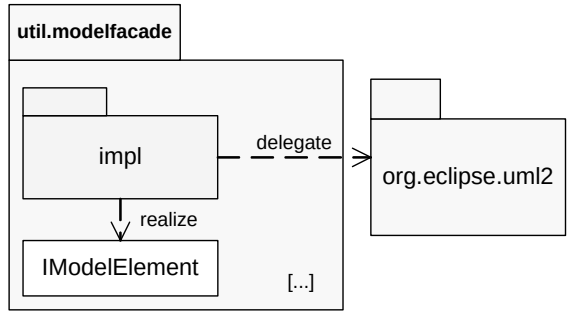


Abb. 9.73 Architekturüberblick Modell-Fassaden

Die Interfaces dieser Fassade werden von genua mandatorisch vorgegeben und sind in `util.modelfacade` zu finden. Grundprinzip ist nun: in den Templates wird nicht direkt auf dem Zwischenmodell (UML2-Objektnetz), sondern rein auf den Interfaces der vorgegebenen Modellfassade gearbeitet und so von der unterliegenden Zwischenrepräsentation abstrahiert. Für ein konkretes Zwischenmo-

dell müssen also dezidierte Implementierungen der Interfaces geschaffen werden. Da genua das `org.eclipse.uml2` Modell zur internen Zwischendarstellung von Modellen benutzt, werden mit dem Framework Wrapper-Klassen ausgeliefert, die Anfragen an die Interfaces der Fassade durch Anwendung des Delegate-Patterns an die Instanzen der UML2-Objekte weiterleiten. Soll ein anderes IR-Modell verwendet werden, muss man sich diese Wrapper selbst schaffen. Abb. 9.73 macht das Vorgehen noch einmal deutlich. Dargestellt ist beispielhaft das Interface `IModelElement`, das neben `IModel` die Basisklasse der Fassade darstellt.

9.6.3 Zusammenspiel der Komponenten

Nun muss noch gezeigt werden, wie die zuvor beschriebenen Emitter-Templates und das Konzept der Modellfassaden in genua gemeinsam eingesetzt werden, um die Generierung von Code zu ermöglichen. Wie wir bereits im Architekturüberblick des Generators gesehen haben, werden die im Ant-Task angegebenen Generatorklassen instanziiert und ihnen als Parameter das Quellmodell übergeben. Zur Erstellung der Artefakte muss nun über die Modell-Elemente des übergebenen Quellmodells iteriert, und dann die in Frage kommenden Elemente in ihre textuelle Repräsentation überführt werden. Abb. 9.74 zeigt dieses Vorgehen am Beispiel der Erzeugung des Quellcodes einer Java-Klasse aus dem zugehörigen Quellmodell.

Im oberen Teil der Abbildung ist der Teil des Templates dargestellt, der vom Java-Generator (Ant-Task) auf dem Gesamt-Quellmodell aufgerufen wird. Generiert werden sollen hier beispielhaft die Java-Klassen. (1) zeigt wie dazu eine Instanz des `Java Class Mapping` Templates erzeugt wird. Die JET-Form dieses Templates stellt der untere Teil der Abbildung dar. Zuerst jedoch weiter im oberen Teil: Mittels einer Factory wird die Modell-Fassade für das übergebenen Quellmodells erzeugt (2), sodass im Folgenden auf dem Interface der zum Modellelement gehörenden Fassade gearbeitet werden kann.

Wie zu sehen ist, stellt `IModel` bereits eine Methode zur Verfügung, die eine lineare Liste aller Elemente des Quellmodells zurückliefert, auf der dann im Weiteren iteriert werden kann. Für alle besuchten Elemente wird wiederum die Fassade erzeugt (3) und für Elemente des Typs `Class` die `generate()`-Methode des zuvor besorgten `Java Class Mapping`s aufgerufen. Diese bekommt

die zuvor erstellte Fassade eines Klassenelementes übergeben, und übernimmt dann die Generation des Quellcodes (4).

Abb. 9.74 Javacode Generierung

```
[...]

<%  Java_Class_Mapping ce = new Java_Class_Mapping();          (1)

    IModel model = (IModel)FacadeFactory.getFacade(argument); (2)
    for (Iterator it1 = model.allModelElementsIterator();
        it1.hasNext();;) {
        IModelElement element =                                (3)
              (IModelElement)FacadeFactory.getFacade(it1.next());
        if (element.isOfType("Class"))%
>
<%= ce.generate(element) %>                                    (4)

[...]                                                uml2javacode.jet
```

```
[...]

<%     IClass clazz = (IClass)argument;                        (5)
%>                                                             (6)
package <%= clazz.containingPackageFullQualifiedName() %>;

<%= clazz.visibility() %> <%= clazz.name() %> {

// DECLARED FIELDS

<% for (Iterator it = clazz.ownedAttributesIterator();
it.hasNext();;) {
   IAttribute attrib = (IAttribute)it.next();
%>
<%= attrib.visibility() %><%= attrib.modifier() %> <%=
attrib.qualifiedType() %> <%= attrib.name() %>;
<% } // end for %>

[...]                                        javaclassmapping.javajet
```

Die übergebene Fassade wird unter dem Namen `clazz` als Implementierung von `IClass` übernommen (5) und wird dann zur Erzeugung der dynamischen Anteile des zu generierenden Textes genutzt. In Bereich (6) sehen wir sowohl die Erzeugung eines statischen Anteils in Form des Schlüsselwortes `package`, gefolgt von der Ermittlung des qualifizierten Packages, dem die Klasse angehört mittels der Fassaden-Funktion `containingPackageFull-QualifiedName()`. An dieser Stelle werden die zuvor theoretisch beschriebenen Vorteile der Fassade praktisch deutlich: Es existiert eine einheitliche Syntax für alle verschiedenen unterliegenden Metamodelle – anstatt des UML2-Frameworks könnte beispielsweise auch eine JMI-basierte Lösung genutzt werden – zudem kann die Syntax der Modellfassade so gewählt werden, dass die Templates möglichst lesbar und damit wartbar bleiben. Der restliche Teil des

Beispiels folgt dem selben Muster und erzeugt die Variablendeklaration der Klasse.

Abschließend bleibt dem Generator-Task nur noch übrig, den erzeugten Quellcode im als Parameter übergebenen Ablageort zu speichern und die Kontrolle an den nächsten Generator zu übergeben. Sind alle Generatoren durchgelaufen, beendet dies den Vorgang der Textgenerierung.

9.6.4 Literaturempfehlungen

Obwohl die vergangenen Abschnitte einen ersten Eindruck von der technischen Realisierung vermittelt haben sollten, stehen sicherlich noch Fragen im Raume, auf die an dieser Stelle aus Platzmangel nicht näher eingegangen werden kann. Im Internet existiert jedoch ein ausführliches Tutorial zum Thema JET, auf das wir an dieser Stelle jeden Interessierten verweisen wollen [Pop04a] und [Pop04b]. Das Fassaden-Prinzip wird auch im AndroMDA Framework [AndroMDA] adressiert. Dort wird als unterliegende Technologie JMI [JMI] unter Zuhilfenahme des Netbeans MDR [MDR] verwendet.

Literatur

[Amb03b] Scott W. Ambler: *Agile Database Techniques*. John Wiley & Sons, 2003.

[Amb04] Scott W. Ambler: *A UML Profile for Data Modeling*. http://www.agiledata.org/essays/umlDataModelingProfile.html (letzter Abruf Mai 2005)

[AndroMDA] AndroMDA.org: *Components quickly with AndroMDA*. http://www.andromda.org/ (letzter Abruf Mai 2005)

[Ant] Apache Software Foundation (ASF): *Apache Ant – Welcome*. http://ant.apache.org/ (letzter Abruf Mai 2005)

[ATL] ATLAS Group: *The ATL home page*. http://www.sciences.univ-nantes.fr/lina/atl/ (letzter Abruf Mai 2005)

[Beehive] Apache Software Foundation (ASF): *Welcome to Apache Beehive*. http://beehive.apache.org (letzter Abruf Januar 2006)

[BHL+04] Jean Bézivin, Slimane Hammoudi, Denivaldo Lopes und Frédéric Jouault: *An Experiment in Mapping Web Services to Implementation Platforms*. http://www.sciences.univ-nantes.fr/info/recherche/Vie/RR/RR-IRIN2004-01.pdf (letzter Abruf Mai 2005)

[CRUD] Wikipedia: *CRUD*. http://en.wikipedia.org/wiki/CRUD_(acronym) (letzter Abruf März 2006)

[EMF] Eclipse.org: *EMF - Eclipse Modeling Framework*. http://www.eclipse.org/emf (letzter Abruf Mai 2005)

[Eva03] Eric Evans: *Domain-Driven Design – Tackling Complexity in the Heart of Software*. Addison-Wesley, 2003.

[JavaCC] java.net: *JavaCC – Documentation Index*. http://javacc.dev.java.net/doc/docindex.html (letzter Abruf Mai 2005)

[JBPM] JBoss Inc.: *JBoss jBPM*. http://www.jboss.com/products/jbpm (letzter Abruf November 2005)

[JMI] Sun Microsystems: *Java Metadata Interface (JMI)*. http://java.sun.com/products/jmi/ (letzter Abruf Mai 2005)

[Jython] jython.org: *Jython Home Page*. http://www.jython.org/ (letzter Abruf Mai 2005)

[JythonAPI] jython.org: *Overview (Jython API documentation)*. http://www.jython.org/docs/javadoc/ (letzter Abruf Mai 2005)

[JythonReg] jython.org: *The Jython Registry*. http://www.jython.org/docs/registry.html (letzter Abruf Mai 2005)

[MDR] NetBeans.org: *Metadata Repository home*. http://mdr.netbeans.org/ (letzter Abruf Mai 2005)

[Nor04a] Theodore S. Norvell: *The JavaCC Tutorial*. http://www.engr.mun.ca/~theo/JavaCC-Tutorial/ (letzter Abruf Mai 2005)

[Nor04b] Theodore S. Norvell: *The JavaCC FAQ*. http://www.engr.mun.ca/~theo/JavaCC-FAQ/ (letzer Abruf Mai 2005)

[Pop04a] Remko Popma: *JET Tutorial Part 1 (Introduction to JET)*. http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html (letzter Abruf Mai 2005)

[Pop04b] Remko Popma: *JET Tutorial Part 2 (Write Code that Writes Code)*. http://www.eclipse.org/articles/Article-JET2/jet_tutorial2.html (letzter Abruf Mai 2005)

[Python] python.org: *Python Programming Language*. http://www.python.org/ (letzter Abruf Mai 2005)

[Ric06] Chris Richardson: *POJOs in Action – Developing Enterprise Applications with Lightweight Frameworks*. Manning, 2006.

[Yacc] compilertools.net: *The Lexx and Yacc page – Yacc*. http://dinosaur.compilertools.net/#yacc (letzter Abruf Mai 2005)

10 /lost+found

„Much good work is lost for the lack of a little more."

Edward H. Harriman

Im Laufe der Erstellung des Buches haben wir viele Themen und Ideen (fremde und eigene) entdeckt, verworfen, unbedingt verwenden wollen und dann doch wieder aus dem Skript herausgenommen, weil wir inzwischen unsere Meinung geändert hatten oder sie einfach nirgendwo „so richtig passen" wollten.

Die Ideen, die wir nach wie vor interessant finden, die jedoch kein eigenes Kapitel füllen, und so vielleicht für immer in der Ablage verschwunden wären, wollen wir sozusagen als „lose Enden" an dieser Stelle zusammenfassen, die zum Weiterrecherchieren anregen sollen.

10.1 Bringt MDA einen ROI? – Die etwas andere Sichtweise.

Sicherlich ist dies die häufigste Frage, die bezüglich einer möglichen Adaption der MDA gestellt wird. Häufigste Reaktion: die Angabe von Studien und Zahlenkolonnen sowie der Versuch, hart messbare Kriterien darüber zu finden, was es bedeutet effizient Software herzustellen. Sind diese Kriterien gefunden, werden Projekte, die mit und ohne MDA-Ansatz durchgeführt wurden untersucht und unter dem Summenstrich die Zahlen verglichen.

Geld ist wichtig, keine Frage. Aus diesem Grund haben wir uns in Kapitel 6 bereits angeschaut, wie eine Rechnung eines MDA-Business-Cases aussehen kann. Auch an dieser Stelle wollen wir der Vollständigkeit halber auf ein Beispiel dieser Art verweisen [TMC03]. An dieser Stelle soll es jedoch vorrangig um einen anderen Aspekt gehen, der bei der Bewertung der Chancen einer MDA-

Einführung eher selten betrachtet wird, unserer Ansicht nach jedoch mindestens genauso wichtig ist.

Neuerungen und Veränderungen des Vorgehens in der Softwareentwicklung – und damit auch die MDA – können nämlich nicht nur an harten Zahlen gemessen werden, sondern auch „weiche" Benefits besitzen, die ebenfalls bei einer Betrachtung des *Für und Wider* berücksichtigt werden müssen.

Vorteil: Spaß an der Arbeit

Unsere These: Qualifizierten Mitarbeitern macht die Arbeit mit den vielen unterschiedlichen Technologien und Themenkomplexen im Dunstkreis der MDA „einfach Spaß". Dies resultiert zum einen aus der tatsächlichen Arbeitserleichterung durch die Anwendung der Prinzipien, die wir im Verlaufe des Buches ausführlich dargelegt haben. Zum anderen jedoch auch in der abwechslungsreicheren Gestaltung, der kreativeren Arbeitsweise und dem kommunikativeren Charakters des Ansatzes *per se*. Daraus können sich unter anderem folgende positive Nebeneffekte für das Unternehmen ergeben:

- eine *Verbesserung des Arbeitsklimas* sowie
- die *Erhöhung der Qualifikation der Mitarbeiter*.

Die Qualifikation der Mitarbeiter wird nahezu „von allein" verbessert und zwar dadurch, dass die initial zu überwindende Lernkurve der MDA gemeistert werden muss. Des Weiteren ergeben sich durch die Notwendigkeit an Anpassungen, die ein MDA-Vorgehen an Struktur und Verantwortlichkeitsverteilung innerhalb der Softwareentwicklung stellt, ganz konkrete neue berufliche sowie persönliche Perspektiven für alle Beteiligten, die neue Impulse liefern können: Rollen bekommen zusätzliche, spannende Akzente bzw. entstehen komplett neu und müssen besetzt werden. Neue Strukturen entwickeln sich und sind willkommener „Frischer Wind". In Projekt- und Akquisesituationen wächst das Reservoir der Bausteine, die zur Lösung der gestellten Aufgaben zur Verfügung stehen. Das Aufzeigen fundierter Alternativen und die Demonstration eines breiten und soliden Know-hows in weiten Themenkomplexen beim Kundengespräch direkt oder durch begleitende Marketingmaßnahmen sichert Wettbewerbsvorteile. Weiterer Positiveffekt: geringere Fluktuation und Vorteile bei der Gewinnung neuer qualifizierter Mitarbeiter.

In Zeiten knapper Ressourcen ist dies ein Benefit, der hoch einzuschätzen ist. Gerade in Zeiten in denen laut Gallup Studie [Gallup05] bereits 18% aller Arbeitnehmer „innerlich gekündigt" haben, da sie sich immer weniger persönlich mit der verrichteten Arbeit identifizieren können und mehr als weitere 69% aus denselben

Gründen nur noch „Dienst nach Vorschrift“ leisten. „Ihrer Arbeit verpflichtet“ fühlen sich sogar nur noch 13%, was sich letztendlich in einem geschätzten gesamtwirtschaftlicher Verlust von ca. 250 Milliarden Euro im Jahr niederschlägt.

Zugegeben: Im Einzelfall können die aufgeführten Punkte nicht im Sinne eines Business Case vorgerechnet werden. Zur generellen Beantwortung der Frage, ob sich eine Beschäftigung mit der MDA lohnt bzw. ob die Investitionen, die eine Entscheidung *für* ein solches Vorgehen ohne Frage nach sich ziehen, für ein Unternehmen gerechtfertigt werden können, sollten jedoch auch die „weichen Faktoren“ nicht unter den Tisch fallen. Die Einführung einer neuen Technologie stellt immer ein Risiko dar, bietet aber im Falle der MDA sowohl *harte* als auch *weiche* Chancen, die den Einsatz ohne Zweifel lohnen. Wie sagt schon Tom DeMarco [DL03]: *„If There's No Risk On Your Next Project, Don't Do It.”*.

10.2 Software-Factories vs. MDA

Anderes Thema: An dieser Stelle wollen wir das Versprechen einlösen, das wir in Kapitel 2 gegeben haben, und einen Ansatz etwas näher beschreiben, der in direkter Konkurrenz zur Model-Driven Architecture entstanden ist: Microsofts *Sofware-Factories*.

Wie bereits beschrieben, wurde der Ansatz von Jack Greenfield [GreenBlog], Keith Short [ShortBlog], Steve Cook [CookBlog] und Stuart Kent [KentBlog] (im Folgenden zur Vereinfachung als „die Vier“ bezeichnet). Niedergeschrieben haben sie ihn im gleichnamigen Buch [GSCK04] sowie den angegebenen Blogs, die einen guten Einstiegspunkt in die leidenschaftlich geführte Diskussion *Software Factories vs. MDA* bieten. Bevor *die Vier* 2002 zu Microsoft wechselten, waren sie allesamt an zahlreichen OMG-Spezifikationen beteiligt, insbesondere an Einreichungen zur UML 2.x, was der genannten Lektüre unschwer zu entnehmen ist. Ihre Vision von moderner Softwareentwicklung wollen sie nunmehr im Produkt *Microsoft Visual Studio Team System* Realität werden lassen [VSTS].

In diesem Abschnitt soll es jedoch nicht um das Werkzeug gehen, das bei ihren Bemühungen im Hause MS entstanden ist, sondern um die Unterschiede der beiden Ansätze und die Betrachtung der Argumente, die gegen die MDA vorgebracht werden, um die Entstehung von *Software-Factories (SF)* als Alternative zu MDA zu rechtfertigen.

Die Informationen der folgenden Abschnitte haben wir dem Papier *Visual Studio 2005 Team System Modeling Strategy and FAQ* [VSTSFAQ] entnommen, dem offiziellen MS Standpunkt zum Thema. Gleichzeitig bildet es auch den Konsens der o. g. *Vier*, weshalb wir uns im Wesentlichen auf dieses eine Papier beziehen und Verweise auf einzelne Diskussionsstränge nur dort angeben, wo Einzelmeinungen und Antworten auf aufgeworfene Fragen von der offiziellen Position abweichen bzw. diese ergänzen.

Beginnen wollen wir mit der offiziellen Definition des Begriffes „Software-Factories". Danach widmen wir uns dem Fokus der „Anti-MDA"-Argumente: der UML. Danach stellen wir die weitere Abgrenzung des Ansatzes zur MDA aus der Microsoft-Sicht vor und wollen im letzten Teil des Abschnittes schließlich weitere Ansichten zum Thema (inklusive unserer eigenen) vorstellen.

10.2.1 Was sind Software-Factories?

Laut FAQ [VSTSFAQ] sind SF „Sets von DSLs zur Erstellung von Systemen aus genau abgegrenzten Familien". Weiterhin „erweitern und umfassen Software-Factories die MDA". Dies bedeutet, dass sich SF nicht auf die reine Angabe von Prinzipien und Standards zur Umsetzung beschränkt, sondern neben der Angabe der Methode auch aus „einer Menge 'Reusable Assets' wie: Modelle, Werkzeuge, Prozesse, Komponenten, Patterns, […]" bestehen, die mit einer Software Factory einer Systemfamilie mitgeliefert werden.

Eine SF pro Systemfamilie

Eine SF korrespondiert dabei mit genau einer dezidierten Systemfamilie und bietet im Einzelnen folgende Assets:

- eine speziell *zugeschnittene Methode* zur Erstellung eines solchen Systems,
- ein *Set von Sichten*, die bestimmte Aspekte des Systems beleuchten,
- die Möglichkeit der *Generierung* von einzelnen (Teil-) Artefakten oder des Gesamtsystems,
- die Möglichkeit der *Validierung* von Artefakten,
- Mechanismen zur Durchführung von *Requirements Engineering*.

10.2.2 Reizwort „UML“

Betrachten wir nun einen der größten Kritikpunkte *der Vier* an der Model-Driven Architecture: dem Vorschlag als Notationssprache die UML zu verwenden. Auch hier wollen wir die Positionen und angeführten Argumente noch nicht bewerten, sondern lediglich unreflektiert wiedergeben.

UML ist nur für Wegwerfmodelle geeignet

Erstes Argument: „Die UML wird von Entwicklern nur zur Erstellung von Sketch-Modellen verwendet.“ Dies sei die Erfahrung von Microsoft als dem Hersteller des am weitesten verbreiteten UML-Modellierungwerkzeugs, die durch Kundenbefragungen gewonnen worden sei. Zu einer solchen „Wegwerf-Modellierung“ lasse sich die UML auch wunderbar verwenden und werde auch von *den Vier* weiterhin zu diesem Zweck empfohlen. Zur weitergehenden Verwendung sei sie jedoch nicht geeignet, vor allem aus folgenden Gründen:

- „Die UML2.x adressiert Aspekte wie DB-Design, Testen, Deployment […] nicht auf intuitive (orig. *engl. natural*) Art und Weise.“
- „Die Definition der *Application Design DSL* [als Notationssprache der SF] mit Mitteln der UML, also Stereotypen und Tags, hätte zu komplexen Modellen basierend auf einer eh schon zu komplexen Spezifikation geführt.“
- „Die Verwendung der Standard UML-Notation für alle Konstrukte unserer [MS] Sprachen hätte die Lesbarkeit und Verständlichkeit der Diagramme beeinträchtigt.“
- „Der UML fehlt Präzision in 'Kernbereichen'.“
- „Das Typsystem der UML passt nicht zu .NET und XML.“

Folglich sei der notwendige Schritt gewesen ein eigenes Metamodell zu schaffen, das die gestellten Anforderungen erfüllt und nicht auf der UML basiert jedoch die Kompatibilität zur MOF wahre und daher alle erstellten Artefakte mittels XMI oder proprietärer Dateiformate mit MOF-basierenden Modellierungswerkzeugen austauschbar seien.

10.2.3 MDA – was fehlt?

Was ist nach Meinung der Vier, neben den Unzulänglichkeiten der UML, also der Haupthinderungsgrund zur erfolgreichen Anwendung der Model-Driven Architecture?

„MDA fehlen Komponenten, die eine erfolgreiche Methodik zur Entwicklung von Software enthalten muss, um ernst genommen zu werden. Vor allem sind folgende Fragen zu beantworten:

- Welche Systeme können entwickelt werden?
- Wie sieht die konkrete Architektur für bestimmte Klassen von Systemen aus?
- Was soll modelliert werden? Hier fehlt die Angabe konkreter Diagrammarten zur Abdeckung der folgenden Themenkomplexe:
  - Erhebung, Analyse und Management von Anforderungen,
  - Modellierung von Aspekten wie Security, Performance und Zuverlässlichkeit […]
  - Packaging und Deployment binärer Komponenten sowie weiterer beteiligter Infrastruktur-Ressourcen und der Zuordnung von Komponenten zu diesen Ressourcen,
  - Modellierung von Testfällen und -daten sowie weiterer Artefakte dieses Umfelds,
  - Modellierung von Aspekten zur Rückverfolgbarkeit (Traceability) von Artefakten und Transformationen
  - sowie Unterstützung zur Modellierung von Aspekten des Konfigurations- und Releasemanagements.
- […]“

Kurz: MDA ist keine Methodik, da ihr wesentliche Bestandteile einer solchen fehlen.

10.2.4 Andere Meinungen zum Thema

Beginnen wir mit den Ressentiments, die *die Vier* gegen die UML hegen. Hier nehmen wir Bezug auf *Microsoft and Domain Specific Languages* [Boo04a] unverändert in unsere Argumentation übernehmen. Dort werden im Einzelnen Argumente aufgenommen und zum Teil widerlegt, die Bezug auf einen Artikel von Alan Cameron Wills (*Why not base domain specific languages on UML?* [Wil04]) sowie auf die Argumente aus *Software Factories* [GSCK04] nehmen und auf die angeblichen technischen Unzulänglichkeiten der UML abzielen. Den Teil des Textes, der sich mit dem Vorwurf beschäftigt, die UML fungiere als „Hammer auch für alle Nicht-Nägel“ und mache es so unmöglich Modelle zu produzieren, die für alle Beteiligten verständlich bleiben, und so den Ansatz der SF begründe nicht-

UML-basierte DSLs zu verwenden, wollen wir ausführlich betrachten:

„[…] the root problem is not simply making one set of stakeholders more expressive, but rather weaving their work into that of all the other stakeholders. This requires common semantics for common tooling and training, so even if you start with a set of pure DSLs, you'll most often end up covering the same semantic ground as the UML.[…]"

Und aus [Boo04b]:

[…] There is no doubt that different domains and different stakeholders are best served by visualizations that best speak their language - the work of Edward Tufte [http://www.edwardtufte.com /tufte] certainly demonstrates that - but there is tremendous value in having a common underlying semantic model for all such stakeholders. Additionally, the UML standard permits different visualizations, so if one follows the path of pure DSLs, you essentially end up having to recreate the UML itself again, which seems a bit silly given the tens of thousand of person-hours already invested in creating the UML as an open, public standard. […]

Hauptziel ist ein einheitliches semantisches Modell

Treffender kann man es kaum ausdrücken: Das Ziel, die Notation möglichst an die dargestellte Domäne anzupassen (DSLs) ist allein für sich genommen die Mühen kaum wert und auch mit UML-Bordmitteln ohne Weiteres zu realisieren. Erst wenn alle erzeugten Modelle der verschiedenen DSLs auch gemeinsam zur Erstellung von Software genutzt werden können, und dies auch automatisiert und werkzeuggestützt passiert, ist ein Hauptziel der derzeitigen Bestrebungen erreicht: eine uneingeschränkte Integration über alle Modellgrenzen hinweg, durch ein einheitliches semantisches Modell. Wenn man dieses Vorhaben jedoch ernsthaft betreibt, landet man früher oder später bei den Aufgaben, die sich die OMG mit den MOF- / UML-Standards auf die Fahnen geschrieben hat. Der vorher so beschworene fundamentale Unterschied in den Zielen von SF und MDA verschwimmt in diesem Aspekt bis zur Unkenntlichkeit. Auch hier gilt es Synergien zu nutzen und nicht wertvolle Energie darauf zu verschwenden das Rad erneut bzw. zweimal zu erfinden.

MDA ist keine Methodik

Auch zum zweiten großen Gegenargument *der Vier*, die „MDA sei keine ernstzunehmende (da unvollständige) Methodik", gibt es eine passende Replik: Wim Bast einer der Submitter der *MOF 2.0 QVT* und Koautor von *MDA-Practice and Promise* [KWB03] stellt in

einem Diskussionsthread auf `TheServerSide.NET` seine Meinung zu diesem Vorwurf wie folgt dar [Bas04]:

„[…] Jack [Greenfield] states that MDA is a methodology and challenges its completeness. He gives a list of important aspects of software development that are not dealt with by MDA in its current state. However, MDA is neither a standard nor a methodology. MDA is a vision on software development. The standards that the members of the OMG develop evolve in a direction that supports that vision. OMG does neither standardize methodologies nor specific implementation technologies. The standards do not dictate all aspects of MDA tools, but only enable interoperability between those tools. OMG standardizes interfaces and languages only. This is for a good reason: The standards give interoperability, but sufficient flexibility to tool vendors to realize their own specific competitive power. It is therefore inappropriate to compare MDA with a complete methodology or a software development tool.
Today the MDA related standards are covering the most important aspects of model driven software development: the exchange of the models by standardizing modeling languages and interchange formats. We do evolve in a direction where more and more can be exchanged. For example, the exchange of transformation rules by standardizing the transformation-rule language. This is in complete agreement with Jack's call for strategy that progressively covers the automation of the software lifecycle. The point is: the tools that realize the software factory approach will not implement standards that are realized by an independent standardization organization, despite the fact that the members of the OMG have been producing those standards for a long time already. Some pretty advanced interoperable MDA tools are on the market for some years now, the software factory approach implementation is just about to make its first babysteps.

More important than the current status is the fact that both approaches have a good chance of succeeding and that they are actually quite in line with each other. In fact, I think that Microsoft's envisioned approach to software factories would fit very well as an implementation of MDA related standards. The issue of platform-independency is the main difference between the visions of Jack's software factories and MDA. Given Microsoft's business model, it is understandable and even reasonable that they are hesitant towards anything associated with platform independency. But like MDA, let's separate business from technology …Even without platform independency MDA has a lot to offer in the area of tool interoperability,

meta-modeling, transformations, etc. Tool vendors that participate in the standardization process of MDA and implement these standards in their tools help their customers to be more successful. The OMG is a democratic organization that gives you enough power to influence the standards if you are willing to put some effort in it. I worked with many other vendors inside the OMG on MDA related standards that where at least as critical as you, Jack. Please feel welcome to translate your valuable criticism on MDA into a contribution to common standards at the OMG. […]"

SF als „Implementierung" der MDA?

Die MDA ist keine Methodik und will auch keine solche sein. Hauptziel ist die Formulierung einer Vision und die Ermöglichung einer Kooperation verschiedener Lösungsansätze durch die Definition von Standards, die ein Zusammenarbeiten und den Austausch von Arbeitsergebnissen ermöglichen. Auch Bast sieht keine fundamentalen Unterschiede der beiden Ansätze, sondern versteht Software Factories als eine mögliche Implementierung der MDA-Vision. Er betont, dass die MDA kein festes Paradigma darstellt, sondern ein gemeinsames Unterfangen, das von Kritik lebt und sich dieser auch stellen kann und will. Dazu ist es allerdings notwendig am „Prozess MDA" aktiv teilnehmen zu wollen und keine proprietären Sonderlösungen zu produzieren.

Nachdem so viele Stimmen zitiert wurden, die anderer Meinung sind als sie die offizielle MS-Sicht der Dinge darstellt, wollen wir mit ausgleichender Gerechtigkeit schließen, und mit einem Zitat aus dem am Anfang des Abschnittes angeführten FAQs [VSTSFAQ] enden:

"Q: Is MS trying to ignore the UML standard and lock customers into their own modeling language? A: No. […]"

Literatur

[Bas04] Wim Bast: *TheServerSide Debates – Software Factories vs. MDA* (Antwort von Wim Bast an Jack Greenfield auf *TheServerSide.net*).
http://www.theserverside.net/news/thread.tss?thread_id=30082 (letzter Abruf Januar 2006)

[Boo04a] Grady Booch: *Microsoft and Domain Specific Languages* (Blog-Eintrag auf *Handbook of Software Architecture*).
http://www.booch.com/architecture/blog.jsp?archive=2004-00.html (letzter Abruf Januar 2006)

[Boo04b] Grady Booch: *Expansion of the UML* (Blog-Eintrag auf *developerWorks : Blogs : Grady Booch*).
www-128.ibm.com/developerworks/blogs/dw_blog.jspa?roll=-20&blog=317 (letzter Abruf Januar 2006)

[CookBlog] Steve Cook: *Steve Cook´s WebLog*.
http://blogs.msdn.com/stevecook/default.aspx (letzter Abruf Januar 2006)

[DL03] Tom DeMarco und Timothy Lister: *Waltzing with Bears – Managing Risk on Software Projects*. Dorset House, 2003.

[Gallup05] The Gallup Organization: *Engagement Index 2005 – Engagement der ArbeitnehmerInnen in Deutschland verharrt auf niedrigem Niveau*.
http://www.presseportal.de/story.htx?nr=719311 (letzter Abruf November 2005)

[GreenBlog] Jack Greenfield: *Jack Greenfield´s Blog*.
http://blogs.msdn.com/jackgr/ (letzter Abruf Januar 2006)

[GSCK04] Jack Greenfield, Keith Short, Steve Cook und Stuart Kent: *Software Factories – Assembling Applications*

with Patterns, Models, Frameworks, and Tools. John Wiley & Sons, 2004.

[KentBlog] Stuart Kent: *stuart kent´s blog*. http://blogs.msdn.com/stuart_kent/ (letzter Abruf Januar 2006)

[KWB03] Anneke Kleppe, Jos Warmer und Wim Bast: *MDA Explained. The Model Driven Architecture – Practice and Promise*. Addison-Wesley, 2003.

[ShortBlog] Keith Short: *Keith Short's WebLog*. http://blogs.msdn.com/keith_short/ (letzter Abruf Januar 2006)

[TMC03] The Middleware Company (TMC): *Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) Approach – Productivity Analysis.* http://www.omg.org/mda/mda_files/MDA_Comparison-TMC_final.pdf (letzter Abruf November 2005)

[VSTS] Microsoft Corporation: *Microsoft Visual Studio Team System*. http://msdn.microsoft.com/vstudio/teamsystem/default.aspx (letzter Abruf Januar 2006)

[VSTSFAQ] Microsoft Corporation: *Visual Studio 2005 Team System Modeling Strategy and FAQ*. http://msdn.microsoft.com/vstudio/default.aspx?pull=/library/en-us/dnvs05/html/vstsmodel.asp (letzter Abruf Januar 2006)

[Wil04] Alan Cameron Wills: *Why not base domain specific languages on UML?* http://blogs.msdn.com/alan_cameron_wills/archive/2004/11/11/255831.aspx (letzter Abruf Januar 2006)

11 Ende gut – alles gut?

„Now this is not the end.
It is not even the beginning of the end.
But it is, perhaps, the end of the beginning."

Winston Churchill

Mit diesem Kapitel sind wir am Ende unseres Streifzugs durch die Technologien und Methoden der Model-Driven Architecture angelangt. Nachzuliefern bleibt noch die Bewertung dessen, was wir in dessen Verlaufe gesehen haben sowie die vorsichtige Einschätzung darüber, wie es mit der Entwicklung der MDA bzw. der Modellgetriebenen Softwareentwicklung im Allgemeinen weitergehen könnte.

Wir wollen damit beginnen darzulegen, was unserer Ansicht nach noch fehlt bzw. was zu tun ist, um die MDA „zum Fliegen zu bringen". Dabei kommen auch die Ansichten weiterer Personen, die sich intensiv mit der MDA beschäftigt haben, zur Sprache, die weitere Ansichten zum Thema liefern. Danach wollen wir einen Blick in die Zukunft wagen und die Frage stellen: „*Wann wird die MDA Commodity?*". In diesem Kontext werden wir auch die Möglichkeit eines Scheiterns sowie die Folgen, die daraus resultieren würden, zur Sprache bringen. Abschließend wollen wir bewerten, ob der Versuch ein „Proof-of-concept" dafür zu erbringen, dass die Durchführung eines MDA-basierten Ansatz zum jetzigen Zeitpunkt, ganz ohne kommerzielle Werkzeuge, durchgeführt und in ein Unternehmen mit bestehendem Softwareentwicklungs-Prozess integriert werden kann, gelungen ist.

Zum Schluss bleibt dann nur noch uns mit letzten dankenden Worten zu verabschieden.

11.1 Was fehlt bzw. ist zu tun?

Beginnen wollen wir mit Betrachtungen zur ersten der genannten Fragen: *„Was ist zu tun?“* bzw. *„Was fehlt zum aktuellen Zeitpunkt noch, um die MDA von einer Cutting-Edge hin zur allgemein anerkannten Best-Practice werden zu lassen?“*.

OMG als Bottleneck

Beginnen wir bei der Suche nach den Gründen für die eher schleppende Verbreitung „echter“ MDA-Lösungen bei den Hütern der MDA selbst: der OMG.

Ein Kritikpunkt: Der derzeitige OMG Standardisierungsprozess ist viel zu langsam. Die OMG möchte Modelle nicht nur zu Softwareentwicklung eingesetzt, sondern sie darüber hinaus zur Verwendung in der GPM sowie im Betrieb, der Wartung und dergleichen mehr verwendet wissen. Sie kommt jedoch mit der Erhebung der Anforderungen an den MDA-Ansatz sowie der Befriedigung des Bedarfs an dementsprechenden Standards nicht in ausreichendem Tempo nach.

So fehlen momentan dringend benötigte Standards was einen großflächigen Einsatz der MDA verhindert. Early Adopters werden so quasi mit der Idee und der UML als Grundgerüst allein gelassen. Will man die MDA jetzt verwenden bleiben einem nur zwei Möglichkeiten: (fast) alles selbst zu tun oder sich auf proprietäre (nicht unbedingt kommerzielle) Lösungen zu verlassen, deren Tragfähigkeit oft nur in einigen Teilbereichen feststeht. Ein Risiko, das sich nur wenige, oft größere Unternehmen leisten können und wollen.

Gleichzeitig wird viel Zeit darauf verwendet, die Werbetrommel zu rühren und damit den Bedarf an konkreten Lösungsvorschlägen zu schüren. Beispiel für einen dringend benötigten und daher lange schmerzlich vermissten Standard ist die QVT (M2M): Vom initialen RFP [QVT-RFP] bis zur finalen Adoption [QVT] vergingen immerhin dreieinhalb Jahre. Ganz ähnlich scheint es derzeit der *MOF Model to Text Transformation Language (M2T)* zu ergehen [M2T-RFP], die zwar in Adveniat ist, ein genaues Datum sich allerdings nicht einmal erahnen lässt. Viele Enthusiasten, akademische Forschungsgruppen und auch nahezu alle kommerziellen Größen machen sich daher daran, die Lücken auf eigene Faust zu schließen, wodurch einem Verwässern und dem damit verbundene Scheitern, der Methode MDA Vorschub geleistet wird (vgl. hierzu die Ausführungen in 11.2.2).

UML – Immer noch verbesserungsfähig

Auch die UML2.x selbst, die als einer der wenigen „MDA-Standards“ in stabiler Form vorliegt lässt noch Wünsche offen. So hält sie zwar für viele Anforderungen statischer und dynamischer Modellierung stimmige und ausreichende Klärungen bereit, bietet sie jedoch immer noch zu viel Interpretationsspielraum und macht so Early Adoptern die Entwicklung eigener Erweiterungen und Auslegungen unumgänglich. Dies hemmt den Standardisierungsgedanken und lässt so einen großen Hebel des Ansatzes nur eingeschränkt wirken. Die bisherige Spezifikation stützt sich auf natürlichsprachliche Formulierungen in Englisch angereichert mit Ausdrücken (Bedingungen/Invarianten) in OCL – hier wäre eine formale Semantik wünschenswert.

Eine solche formale Fixierung der Semantik ließe sich beispielsweise im Sinne einer Übersetzersemantik über entsprechende Transformationsregeln erfassen. Wie wir gerade gelesen haben ist über die QVT mittlerweile ein standardisiertes Vehikel zur Beschreibung solcher Regeln verfügbar. Ob die OMG in Zukunft allerdings auch standardisierte Transformationsregeln damit liefern wird, oder ob sich beispielsweise für die Abbildung von UML nach Java der Java Community Process (JCP) darum kümmert, oder ob wir gar weiterhin standardlos in dieser Hinsicht leben müssen, steht bislang noch in den Sternen. Gerade in Bezug auf den nachfolgend zu diskutierenden Punkt der *Fehlenden Experten* scheint uns aber eine einheitliche Ausgangsbasis zum Erlernen der neuen Techniken als unerlässlich. Zugleich muss diese Basis aber offen und flexibel anpassbar bleiben, damit der durch die modellgetriebene Herangehensweise gewonnene Nutzen nicht gleich wieder hinfällig wird.

Fehlende Experten

Zu den Risiken, die ein Unternehmen auf sich nehmen muss, das heute einen modellgetriebenen Ansatz fahren möchte, gehört auch, dass es dies oft ohne Experten in den eigenen Reihen tun muss.

Wie in einer Studie der Middleware Company [TMC03] treffend bemerkt wird:

„It [*the MDA*] *makes brain surgeons better brain surgeons, but it won’t make janitors into brain surgeons.”*

Folge: Die zugegebenermaßen steile Lernkurve zur Anwendung der Methoden und Techniken muss durch *Learning by doing* (=hohes Risiko + Kosten durch (vorübergehend) verminderte Produktivität/Effizienz) oder durch die Inanspruchnahme von *Coaching* (ver-

mindertes Risiko + Kostenfaktor Coaches) bewältigt werden. Beide Alternativen klingen beim erstmaligen Hören in Managerohren oft wenig „sexy“, was oft in einem Verharren in etablierten Strukturen und Prozessen führt.

Dieses Verharren wird noch bestärkt durch das momentane immense Angebot an Java-, C++- oder auch C#-Programmierern , das sicherlich in der guten Dokumentation dieser Sprachen sowie den hohe Ausbildungskapazitäten begründet ist, und dem gleichzeitigen Mangel an höherqualifiziertem Personal in Feldern der MDA. Gründe den Wechsel dennoch zu wagen gibt es genug, und wurden auch in diesem Buch bereits zu genüge angeführt, sodass wir sie an dieser Stelle nicht noch einmal wiederholen wollen.

11.2 Was wird?

Wagen wir nun einen vorsichtigen Blick in die Zukunft und beschäftigen uns mit den beiden Hauptmöglichkeiten, die die Zukunft der MDA in der Softwareentwicklung bilden.

Im ersten Fall wollen wir dabei davon ausgehen, dass sich die MDA unter den vielen Strömungen der modellgetriebenen Ansätze behaupten kann, während wir in der zweiten Betrachtung einige Gründe vorstellen, die unserer Meinung nach auch jetzt noch zu einem Scheitern des Ansatzes führen können.

11.2.1 Wie lange dauert es noch bis MDA zur Commodity wird?

Gehen wir zuerst also davon aus, dass die Model-Driven Architecture sich durchsetzt, und tatsächlich zu einer weit verbreiteten Vorgehensweise wird, wie Software in Zukunft entwickelt wird. Wann entwächst MDA der *Cutting-Edge* und wird für jedermann kostengünstig und effizient sowie mit niedrigem Risiko verfügbar? *Wann wird MDA zur Commodity?*

Ein Gedankengang, der sich mit diesem Thema beschäftig, und den wir an dieser Stelle vorstellen wollen, stammt von Jean Bézivin und basiert auf einem Vortrag namens *MDA™ From Hype to Hope and Reality* [Béz03], den er auf der «UML 2003»-Konferenz gehalten hat.

Die Annahme, die hier getroffen wird, ist die, dass sich die Technologie MDA mit der Idee der Objektorientierung (OO) vergleichen lässt, die Anfang der 80er Jahre (des letzten Jahrhunderts ;o)) aufkam und deren Verfechter sich mit denselben initialen Problemen

und Bedenken auseinandersetzen mussten, wie es die Protagonisten der MDA jetzt tun. Die Kernaussage der Beobachtung ist die Folgende:

Früher war alles ein Objekt, heute ist alles ein Modell (orig.: From "Everything is an object" to "Everything is a model").

Die Modellorientierung setzt den Trend der Softwareentwicklung zu zunehmender Abstraktion fort und löst damit die Objektorientierung als „aktuellstes" Paradigma ab. Viele Lehren, die bei der Umsetzung der „Idee Objektorientierung" gemacht wurden, können und sollten, bei der Ausarbeitung und Vervollkommnung der MDA genutzt werden und nicht dieselben Fehler noch einmal unnütz wiederholen (vgl. dazu die Ausführungen zur *Wiedergeburt der Methodenkriege* in Abschnitt 11.2.2)

Geht man also davon aus, dass sich OO und MDA in wesentlichen Punkten gleichen, kann man aus der Zeit und den Phasen die das OO-Paradigma durchlaufen hat auf die Zeit schließen, die die MDA wahrscheinlich braucht, um zur ausgereiften Vorgehensweise mit umfassender technologischer Unterstützung zu werden? Wenn man es versucht, ergibt sich folgendes Bild (Abb. 11.1):

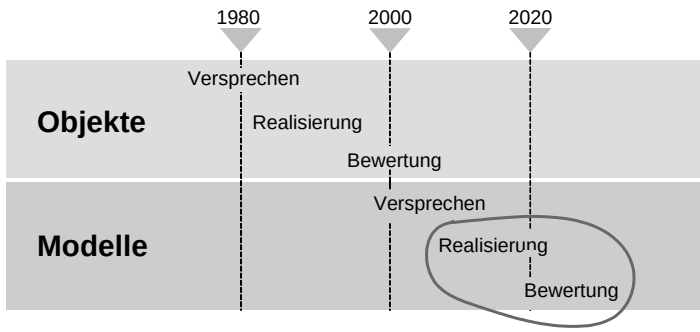


Abb. 11.1 OO, MDA und die Zeit

Betrachtet wird der Zeitraum, in dem es ein Paradigma vom Versprechen (der Vision) über die Realisierung, bis hin zur Bewertung und Weiterentwicklung schafft und schließlich seinem Nachfolger als Fortführer des Trends Platz macht.

Betrachtet man diese Phasen für die OO ergibt sich folgendes Bild: Nach ihrem Auftreten in den frühen 80er Jahren und den beschriebenen „Kriegen" um die beste Umsetzung der Idee, begann sie in den 90ern zum allgemein anerkannten Vorgehen zu werden, bis sie ab Mitte der 90er zum *State-of-the-art* Vorgehen wurde und bis zum heutigen Zeitpunkt fast alle anderen Vorgehen in Nischen ver-

drängt hat (zumindest in der Domäne der Enterprise-Anwendungen, auf die wir bei unseren Betrachtungen bisher den Schwerpunkt gelegt haben). Auf ihrem Höhepunkt wurden allerdings auch erste kritische Stimmen laut, die anmerkten, dass auch die OO nicht alle Versprechen hatte halten können, die ursprünglich von ihr erhofft wurden (man denke zum Beispiel an das Auseinanderklaffen der versprochenen Einsparungen durch die Wiederverwendung von Objekten und der harten Realität).

Eine neue Phase im Lebenszyklus des Paradigmas OO begann: Die Umsetzung der Vision war vollendet und die Bewertung der Erfolge begann. Nun wurden Probleme des Ansatzes identifiziert und nach Möglichkeiten der Weiterentwicklung geforscht. Alles in allem der Weg, der seit Beginn des Software-Engineering beschritten wird, um zu Fortschritten in der Entwicklung zu gelangen: Fehler des alten Vorgehens identifizieren und Verbesserungen oder Neuerungen dort anbringen, wo die vorhandenen Mittel nicht reichen.

Im Falle der OO umfasst dies einen Zeitraum von gut zwanzig Jahren. Gehen wir davon aus, dass sich die MDA zurzeit in der Phase der Realisierung befindet (markiertes Gebiet der Abbildung) haben wir also noch gut fünf Jahre vor uns, bis die MDA dort ist, wo die Objektorientierung Mitte der 90er war: der Hype klingt ab und erste Erfahrungen mit der Anwendung der Idee in produktivem Umfeld werden gemacht.

Was bedeutet das für die Unternehmen?

Für alle Unternehmen, die jetzt in die Umsetzung einsteigen, besteht noch die Chance, „von Anfang an mit dabei zu sein“ und sich so einen Wissensvorsprung und damit einen Wettbewerbsvorteil zu sichern. Bei zu abwartender Haltung besteht jedoch die Gefahr den Trend zu verpassen, mit den entsprechenden negativen Folgen, die daraus entstehen.

Mit fortschreitender Reife der MDA verringert sich das Risiko, allerdings werden auch die Früchte kleiner, die man mit der Umsetzung der Prinzipien und Ideen ernten kann. Eines macht die Zeitspanne auf jeden Fall klar: Wer jetzt beginnen möchte, muss selbst aktiv werden, da die Selbstverständlichkeit, mit der wir heute die Objektorientierung verwenden, für die MDA noch in weiter Ferne ist.

11.2.2 Woran könnte die MDA noch scheitern?

Dass ein frühzeitiges Aktivwerden in der Umsetzung der MDA im eigenen Unternehmen die von uns empfohlene Strategie ist, da sie

die unserer Meinung nach konsequente und notwendige Fortführung des Trends im Software Engineering zur Abstraktion darstellt, und momentan ohne vernünftige Alternativen ist, sollte im Verlaufe des Buches deutlich geworden sein. Trotzdem besteht die Gefahr, dass sich zumindest die Model-Driven Architecture als eine Methodik unter vielen nicht durchsetzt und im allgemeinen „Kampf um das beste Kürzel“ untergeht.

Die Hauptgefahren

Ein Scheitern droht der MDA unserer Meinung nach vor allem in folgenden Hauptgefahren: *Fehlender Anstand* der *Wiedergeburt der Methodenkriege* und *Zu langer Time to Market.*

Fehlender Anstand

Unter dem etwas pathetischen Begriff sehen wir alle Aspekte, die mit dem Missbrauch der „Idee MDA“ zu Zwecken, die nichts mit der Weiterentwicklung der Softwareentwicklung, sondern nur zur Erlangung von Vorteilen persönlicher oder kommerzieller Art zu tun haben. Beispiele hierfür sind:

- *Ein Ausverkauf der Idee.* Der MDA droht dasselbe Schicksal, wie einst den Case-Tools, die auch vor ihrer Reife mit allzu vollmundigen Versprechungen an den Mann gebracht wurden, was die Enttäuschung über den tatsächlichen Stand der Technik um so herber hervortreten ließ.
- *Unehrlichkeit über den Stand der Dinge.* Die MDA befindet sich noch in ihren Kinderschuhen und ist lange davon entfernt ausgereift, einfach und ohne großes Risiko von jedermann im vollen Umfang verwendet werden zu können. Um trotzdem auf den Zug aufzuspringen wird dies von Toolherstellern und Beratern gleichermaßen nur zu gerne verheimlicht. Dies bedeutet nicht, dass es generell zu früh ist für die Anwendung der MDA. Man muss jedoch ehrlich sein und auch Situationen identifizieren, für die die Zeit (noch) nicht gekommen ist. Sonst droht auch hier der Vertrauensverlust und damit die „Verteufelung“ der Gesamtidee.
- *Vorgeben alles sei einfach und unter Kontrolle.* Auch hier die Warnung vor drohendem Vertrauensverlust. Das Themengebiet MDA ist vielfältig, unübersichtlich und auch von den gestellten Aufgaben und Herausforderungen eines der komplexesten des Software Engineering. Kenntnis und Zusammenspiel von theoretischer Informatik, harter Programmierung sowie Projektmanagement und Methodik machen eine Umsetzung nicht zu etwas, „was man so nebenbei erledigt.“ Zusätzlich sind viele

Entwicklungen, wie die Dauer von Standardisierungen und deren Umsetzung in verwendbare Werkzeuge nicht genau vorhersagbar, was das Risiko einer *Early Adoption* noch erhöht. Noch einmal: Risiken sind Chancen und können auch so kommuniziert werden. Zu Chancen werden sie jedoch erst dann, wenn sie zuvor als Risiko erkannt und behandelt werden.

Wiedergeburt der Methodenkriege

Im Verlaufe des Buches und dieses Kapitels wurde bereits auf die Gemeinsamkeiten der Grundlagen der MDA und der Objektorientierung hingewiesen bzw. die MDA als konsequente Fortführung der Konzepte verstanden, die über die letzten 20 Jahre durch die OO entdeckt und fundiert worden sind.

Im Sinne dieses „Lernens aus der Historie" sollte natürlich auch die Wiederholung von bereits gemachten Fehlern vermieden werden, Theorie und Realität klaffen jedoch auch hier, wie so oft, auseinander. Ähnlich wie zu Zeiten der sogenannten „Methodenkriege" (engl. method wars) Anfang der 90er schicken sich einige Gruppen dazu an *ihre* Interpretation der MDA als die einzig Wahre zu verstehen und keine anderen Nebenbuhler neben sich zu dulden.

Ob Model-Driven Architecture (MDA), Modellgetriebene-Softwareentwicklung (MGSE) oder „Software, Modelle, Entwicklung" (SME) oder … Wie bereits dargelegt, ist letztlich alles Teil des gleichen, aus der Notwendigkeit heraus geborenen Trends des Software Engineering mit den selben zugrunde liegenden Basiskonzepten. Haarspaltereien, welches Kürzel nun treffender ist, sind oft Selbstzweck und bei näherer Betrachtung fehlen nahezu immer die angegebenen gravierenden Gründe warum Ansatz XY eben *nicht* MDA ist (vgl. hierzu die Ausführungen zu Software-Factories in Kapitel 10). „Fast MDA – nur anders." Eine Menge solcher Ansätze existieren zurzeit vor allem im „agilen Fahrwasser". Agil bedeutet hier schlicht, dass Aspekte der MDA, die momentan nur schwer umzusetzen sind einfach weggelassen werden und dies als „agiles Vorgehen" entschuldigt wird.

Ein wichtiges Beispiel der aktuellen Entwicklung ist die M2M-Transformation. Hier fehlen Standards und die Umsetzung in konkrete Tools ist aufwendig und schwierig. Viele Frameworks lassen sie daher komplett weg und versuchen zusätzlich diese Entscheidung mit der Behauptung: „eine PIM-PSM Transformation sei sowieso nicht notwendig" sie nachträglich mit einem mehr oder weniger überzeugend klingenden Unterbau zu versehen. Ein solches Vorgehen ist unserer Meinung nach allerdings weder sinnvoll noch zielführend. Ziel muss es sein alle vorgeschlagenen Aspekte der MDA aus-

zuprobieren und erst dann zu entscheiden was sich bewährt hat und was nicht.

Um es noch einmal platt zu wiederholen: MDA ist nichts revolutionär Neues, sondern bündelt mit ihrem Namen eine Menge bekannter Ansätze und Standards, die das Leben in der SE bewiesenermaßen vereinfachen. Die Bestandteile der MDA sind ferner alle *kein* „Muss“. Wo Ergänzungen tatsächlich gebraucht werden sollten diese in die Diskussion eingebracht und so die MDA als standardisiertes Vorgehen weiterentwickelt werden.

MDA kostet noch nicht einmal etwas, also: Warum nicht MDA? Wenn nun also in der Art argumentiert wird „Mein Kürzel ist besser als Deins.“ sollte generell die Alarmglocke schellen: Warum ist es wichtig, wie etwas heißt? Welche Interessen verbergen sich dahinter? Ist es tatsächlich wichtig ob MDA oder MGSE?

Was momentan gebraucht wird, sind weniger Diskussionen der Art „Wer treibt wen?“ oder „Wessen Kürzel ist schöner oder war eher da?“, sondern Projekte, die den Gedanken verwirklichen, der allen (angeblich) unterschiedlichen Vorgehen gemeinsam ist. Kann man sich wirklich nicht darauf einigen diesen Gedanken *MDA* zu nennen? Auch die Objektorientierung hat ihren Durchbruch erst geschafft, nachdem alle Kräfte gebündelt wurden um die hehren Ziele zu verwirklichen. Wie wir im Verlauf dieses Kapitels gesehen haben, hat es trotzdem einige Zeit gedauert, bis alle Vorteile der OO der Allgemeinheit als Asset zur Verfügung standen. Es sollte also vermieden werden in endlosen Grundsatzdebatten ohne Substanz den Hauptantrieb zu zerfasern und am Ziel einer standardisierten Realisierung festzuhalten … nennen wir sie doch MDA.

(Anm.: Zum Themenkomplex „Warum das Rad doppelt erfinden?“ haben wir bereits in 10.2.4 interessante Meinungen weiterer Befürworter der MDA vorgestellt. Die dort aufgeführten Bemerkungen zum Thema UML gelten in generalisierter Form auch an dieser Stelle unverändert.)

Time to Market

Als letzten wichtigen Punkt, der quasi die Gretchenfrage für jedes neue „Paradigma” darstellt, wollen wir noch die Zeit anführen, die es dauert, bis erste Umsetzungen der MDA sich in Realprojekten und in kommerzieller Toolunterstützung manifestieren, und damit das Fundament der MDA als anerkannte Best-Practice festigen. Geschieht dies nicht oder zu spät, da zu viel Zeit in Diskussionen um den richtigen Namen und zu wenig Einsatz zur Weiterentwicklung

und Verbesserung der ersten Vorstellungen verwendet werden, kommen andere Ansätze, die sich an Einzelteilen der MDA bedienen, und dann mehr oder minder erfolgreich eingesetzt werden, was letztendlich den Versuch der Vereinigung dieser Einzeltechniken zu einem Gesamtvorgehen zunichte machen. [Béz03]:

If the MDA does not fly, then other technological spaces will harbor these ideas.

Der erfolgreiche Einsatz in Einzelteilen ist grundsätzlich natürlich nichts Schlimmes. Wie wir in der Betrachtung der Herkunft der MDA gesehen haben, wurden viele der Ursprünge ja auch bereits erfolgreich eingesetzt. Im aktuellen Vorhaben geht es jedoch um mehr: um das Verschmelzen der Einzeltechniken zu einem Gesamtgebilde – dem „Paradigma" MDA. Hier besteht die Chance Energien zu bündeln und so die Schwächen auszugleichen, die den einzelnen Teilen bisher den großen Durchbruch verwehrt haben.

11.3 Proof-of-Concept erfolgreich?

Nach allen Betrachtungen zu theoretischen, praktischen und teilweise auch mehr oder weniger ideologischen Aspekten der Model-Driven Architecture, wollen wir nun versuchen ein Fazit des Stands der Dinge in Sachen MDA zu ziehen.

Am Anfang des Buches hatten wir das Ziel formuliert mit dieser Publikation so etwas wie ein *Proof-of-Concept* der MDA zum jetzigen Zeitpunkt zu versuchen. Erreicht werden sollte die Konzeption eines prototypischen Frameworks zur Realisierung der MDA-Vision sowie die Einbettung in einen Softwareprozess, wie er üblicherweise heutzutage realen Projektsituationen entspricht.

Sicherlich haben Sie sich zu diesem Zeitpunkt bereits selbst ein Bild darüber gemacht, ob der Versuch gelungen ist. Im Folgenden wollen wir die Ergebnisse aus unserer Sicht noch einmal kurz zusammenfassen.

11.3.1 Der MDA-Prozess – in Sicht?

Nach der Beleuchtung der theoretischen und konzeptionellen Grundlagen, stand als erstes die Auswahl eines geeigneten Prozessmodells zur Unterstützung unseres fiktiven MDA-Projekts auf dem Plan. Nach anfänglicher Ratlosigkeit – die MDA schlägt ja explizit keine dezidierte Methodik vor –fällt schnell die Ähnlichkeit des konzeptuellen Vorgehens mit den notwendigen Aktivitäten und Ergebnissen innerhalb der Welt der Software Produktlinien auf. Hier existieren

bereits bewährte Prozessmodelle (Beispiel FAST [WeLa99]), die als Muster bei einer Adaption dienen können und uns bei den beschriebenen Besonderheiten eines MDA-Prozesses als Grundlage gedient haben. Weiterhin fällt grundsätzlich die Affinität zur Iterativ-Inkrementellen Softwareentwicklung (IID) auf, die mittlerweile zur de-facto Standard-Basis moderner Prozessmodelle geworden ist, und nahezu ideal für die Anforderungen einer modell-basierten Vorgehensweise geeignet ist.

Gefragt, ob zurzeit ein tragfähiges Modell zur Durchführung von MDA-Prozessen existiert können wir daher ruhigen Gewissens mit „Ja." antworten und von der Machbarkeit einer Adaption ausgehen.

11.3.2 genua – Prototypisches MDA-Framework

Im zweiten Teil des Buches wurden dann die kritischen Funktionalitäten, die eine MDA Infrastruktur bieten muss identifiziert und mit Open-Source Werkzeugen prototypisch umgesetzt. Abb. 11.2 zeigt einen Überblick der geschaffenen Komponenten, in Form der entwickelten Eclipse-Plugins.

Realisiert wurden prototypisch:

- Ein UML-Editor (Klassen- und Aktivitätsdiagramme), der auch die Erstellung und Anwendung von Profilen ermöglicht.

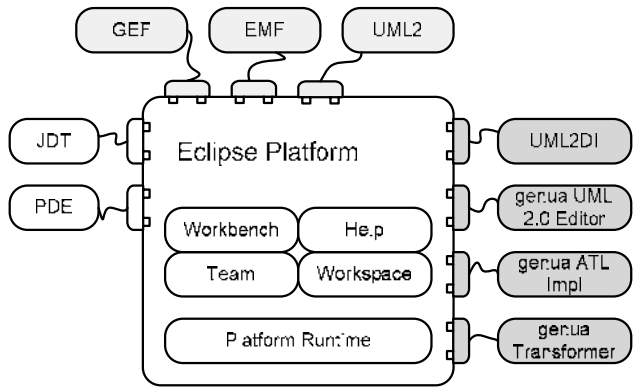


Abb. 11.2 genua – Plugin Architektur

- Eine deklarative Transformationsbeschreibungssprache (gATL) zur Beschreibung von Modell-zu-Modell Tranformationen.

- Ein Transformator, der sowohl die M2M-Transformation mittels gATL, als auch die M2T-Transformation (Codegenerierung) unterstützt.
- Plattformunabhängige UML-Profile zur Darstellung von Persistenz- sowie Dialog- und Workflow-Aspekten sowie die zugehörigen PSMs in Form von Hibernate, Beehive, jBPM und JavaServerFaces Profilen.
- Die Templates zur Generierung textueller Artefakten (Code, Deskriptoren, …) aus den o. g. PSMs.

Die benutzten Technologien kommen alle aus dem Open-Source-Bereich und sind daher keinesfalls an sich kostspielig. Ist man zudem gewillt in die praktische Umsetzung zu investieren und diese selbst in die Hand zu nehmen und Fragen direkt an die OS-Projekte zu stellen bekommt man auch schnell Antwort, die oft die Lösungen für auftauchende Probleme darstellt, und die Eigenimplementierung eines MDA-Frameworks in erreichbare Nähe rücken lässt. Der Umgang mit dem Framework, integriert in eine IDE wie Eclipse ist komfortabel und lässt vermuten, dass die Benutzerakzeptanz nach einigen Vorbehalten auch in der Breite der IT nicht lange auf sich warten lassen wird.

11.4 … schließende Worte

Den folgenden Personen und Institutionen gilt unser Dank:

- Astrid Baumgart und ihrem Team, unser orthografisches Gewissen, für die umfangreiche Unterstützung bei der Erstellung des Manuskriptes,
- der adesso AG für die finanzielle Unterstützung bei der Erstellung dieses Buches,
- Carsten Brand, „unserem“ Versicherungsexperten, für die Unterstützung bei der fachlichen Erarbeitung des Fallbeispiels auch in späten Abendstunden,
- dem oose.de-Team – insbesondere Christian Weiss – für fünf Tage *OOGPM unplugged* im schönen Hamburg,
- unseren Damen – Petra und Iris – die an zahllosen Wochenenden auf uns verzichten mussten (?), an denen ~~unser Baby~~ Buch an erster Stelle stand

Bis zum nächsten Buch!

Literatur

[Béz03] Jean Bézivin: *MDA™ From Hype to Hope and Reality*. In: «UML» 2003, San Francisco, Invited Talk. http://www.sciences.univ-nantes.fr/info/perso/permanents/bezivin/UML.2003/UML.SF.JB.GT.ppt (letzter Abruf Mai 2005)

[M2T-RFP] Object Management Group (OMG): *MOF Model to Text Transformation Language – Request For Proposal*. http://www.omg.org/docs/ad/04-04-07.pdf

[QVT] Object Management Group (OMG): *MOF QVT Final Adopted Specification*. http://www.omg.org/docs/ptc/05-11-01.pdf (letzter Abruf November 2005)

[QVT-RFP] Object Management Group (OMG): *MOF 2.0 Query / Views / Transformations RFP*. http://www.omg.org/docs/ad/02-04-10.pdf (letzter Abruf Mai 2005)

[TMC03] The Middleware Company (TMC): *Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) Approach – Productivity Analysis*. http://www.omg.org/mda/mda_files/MDA_Comparison-TMC_final.pdf (letzter Abruf November 2005)

[WeLa99] David M. Weiss und Chi Tau Robert Lai: *Software Product-Line Engineering – A Family-Based Software Development Process*. Addison-Wesley, 1999.

A UML-Schnellreferenz

Der vorliegende Anhang liefert eine kurze Einführung in die aktuell insgesamt dreizehn Diagrammtypen der UML 2.x. Aufgrund der Menge an Konzepten und Notationselementen müssen wir uns allerdings bei der Beschreibung auf das Notwendigste beschränken. Vor allem Elemente, die im Fallbeispiel des Buches eine besondere Rolle spielen, werden genauer betrachtet. Für eine detaillierte Beschreibung aller Elemente verweisen wir jedoch auf die in A.3 genannten Literaturtipps am Ende des Anhangs.

A.1 Strukturdiagramme

Beginnen wir mit der Kategorie Diagramme, die die statischen Eigenschaften des modellierten Systems in den Vordergrund stellen: den Strukturdiagrammen. Die UML bietet hier aktuell sechs verschiedene an, die wir im Folgenden näher betrachten wollen.

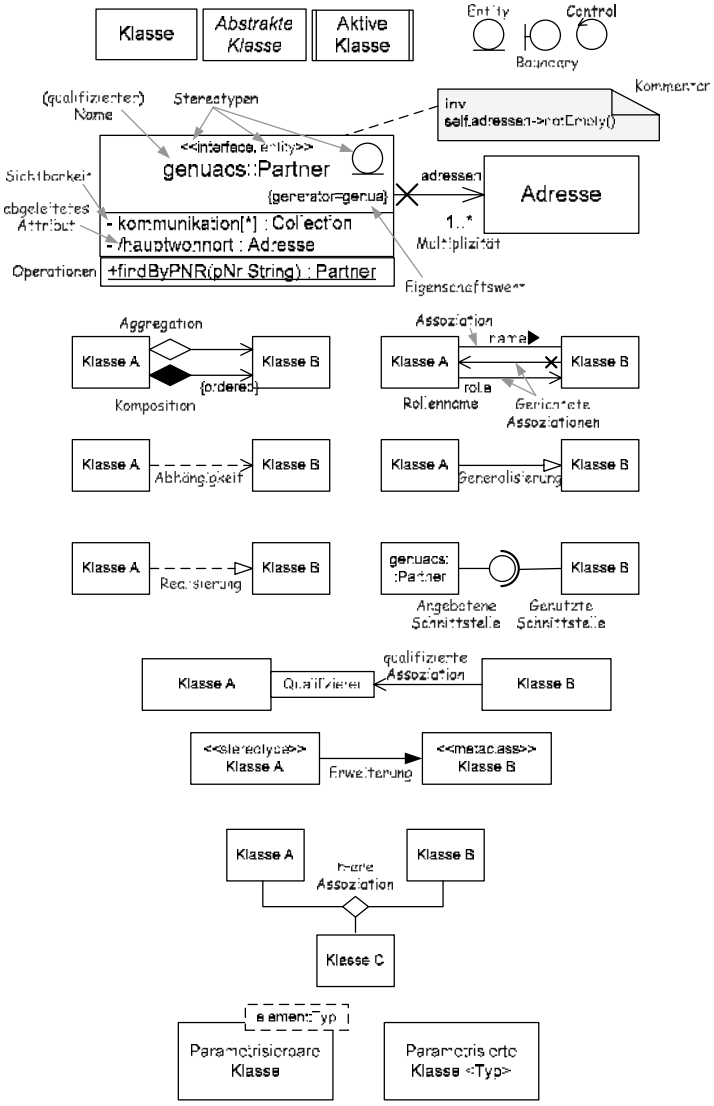
A.1.1 Klassendiagramm

Das bei weitem bekannteste Diagramm des UML-Stacks stellt das Klassendiagramm dar, das zur Darstellung der statischen Struktur eines Systems dient. Dabei können sowohl fachliche Aspekte modelliert werden, als auch konkrete Datenstrukturen.
Kommen wir nun zu einer Kurzbeschreibung der in Abb. A.1 dargestellten Notationen:

Statische Eigenschaften

Im oberen Teil sehen wir das Standardsymbol für *Klassen und Objekte* in der UML – das Rechteck – in dessen Mitte der Klassenname eingetragen wird. Namen in Kursivschrift bedeuten, dass es sich um eine abstrakte Klasse handelt, die erst von einer Subklasse ganz oder in Teilen vervollständigt werden muss, bevor sie instanziiert werden kann. Alternativ zur Kursivschreibung kann neben oder unter dem Klassennamen auch der *Schlüsselwort (engl. keyword)* {`abstract`} notiert werden (hier nicht dargestellt).

Abb. A.1 Notationen im Klassendiagramm



Syntax von Attributen

```
[ +|# -| ~| /] Name : [Multiplizität Ordnung] [ =Initialwert] [{Eigenschaftswerte}]
```

Syntax von Operationen

```
[ +|# -] Name ([<Parameterart> ] Name : [Multiplizität] [{Eigenschaftswerte}]
<Parameterart> ::= [<Richtung>] Name [: Typ] [ =Initialwert]
<Richtung> ::= in | out | inout
```

Element-Namen müssen innerhalb des umgebenden Namensraums – in der UML durch *Packages (Pakete)* realisiert – eindeutig sein (vgl. auch A.1.3). Setzt man vor den Klassennamen die Hierarchie der

Pakete, in denen die Klasse enthalten ist (Beispiel: `java::util::List`), kann so auch auf Elemente zugegriffen werden die außerhalb des aktuellen Namensraums liegen, ohne sie zuvor explizit importieren zu müssen. Die Angabe der Pakethierarchie geschieht von außen nach innen. Für unser Beispiel bedeutet das: `java` ist das äußerste Paket, `util` liegt innerhalb `java` und `List` ist in `util` selbst lokalisiert. Namen, denen ihre *Containment-Hierarchie* als Präfix vorangestellt sind, nennt man *voll-qualifiziert (engl. full qualified)*.

Unterteilung von Klassen durch Compartments

Klassen repräsentierende Rechtecke können in beliebig viele weitere Sinnabschnitte unterteilt werden. Einzige Vorgabe der UML ist, dass im obersten dieser Abschnitte *(engl. compartments)* der Name des Elements zu finden sein muss. Der Inhalt der übrigen Abschnitte kann nach eigenen Kriterien (beispielsweise Aufgabenbereiche) unterteilt werden. Als Quasi-Standard hat sich die 3er-Teilung in Namens-, Attribut- und Operationskompartment durchgesetzt, die auch in allen gängigen Modellierungstools unterstützt wird.

Attributen und Operationen kann ihre *Sichtbarkeit* vorangestellt werden. Da Sichtbarkeiten bzw. die Kenntnis über deren korrekten Gebrauch bei zunehmend formalisierter Nutzung der UML eine wichtige Rolle spielen. und uns mittlerweile auch einige falsche Darstellungen begegnet sind, soll die folgende Tabelle kurz die möglichen Notationen und ihre Bedeutung veranschaulichen:

Abb. A.2 Sichtbarkeiten Übersicht

UML Notation	Name	Bedeutung
+	`public`	unbeschränkter Zugriff im System
-	`private`	nur Zugriff innerhalb der umgebenden Klasse
#	`protected`	Zugriff innerhalb des umgebenden Pakets und aller Subklassen der umgebenden Subklasse
~	`package`	Zugriff nur innerhalb des umgebenden Pakets

Erwähnenswert ist der oft unbeachtete Unterschied zwischen `protected` und `package`, da die wesentlich restriktiveren Einschränkungen des `package` Modifiers, der in Java defaultmäßig verwendet wird falls keine Sichtbarkeit explizit angegeben ist, zu zuerst scheinbar rätselhaften Problemen bei ersten Gehversuchen in formaler Modellierung und anschließender Codegenerierung führen kann.

Abgeleitete Elemente

Abgeleitete Elemente *(engl. derived elements)* werden durch Voranstellung eines Slashes („/“) vor den Elementnamen gekennzeichnet und finden in den Spezifikationen der OMG rege Verwendung. Per definitionem sind abgeleitete Elemente solche, die dem Modell selbst keine zusätzliche Information hinzufügen, sondern aus bereits vorhandenen Modellelementen berechnet (mit anderen Worten *abgeleitet*) werden können. In Abb. A.1 ist dieser Sachverhalt beim Attribut `hauptwohnort` gegeben. Hier handelt es sich um eine Adresse, die bereits im Attribut `adressen` enthalten ist. Die Adresse `hauptwohnort` muss also nicht explizit gespeichert werden, sondern dient lediglich als besondere Referenz auf einen bereits vorhandenen Wert innerhalb dieser Liste von Adressen. Der konkreter Wert kann also abgeleitet werden. Die eventuelle Notwendigkeit in der konkreten Implementierung des Systems den Wert trotzdem innerhalb redundanter Datenstrukturen vorhalten zu müssen, um nicht beispielsweise unnötige rekursive Suche o. Ä. nötig zu machen, die dann in inakzeptablen Laufzeiten der Applikation resultieren, bleiben davon völlig unberührt. Auf diesem Abstraktionsniveau liegt der Fokus rein auf der Darstellung der Berechnungsmöglichkeit.

Neben der *Multiplizität*, die wir im Zusammenhang mit den im Folgenden betrachteten Assoziationen noch näher betrachten wollen, kann noch ein *Initialwert* des Attributes angegeben werden sowie wie für jedes Element noch eine Liste von Schlüsselworten. Werden Namen von Attributen oder Operationen unterstrichen handelt es sich um *statische Elemente (synonym: Klassenelemente)*. Diese Elemente (und deren konkrete Werte) sind nicht instanzspezifisch, sondern gelten übergreifend für alle Instanzen einer Klasse. So muss zum Zugriff auf die Methode `findByPNR(pNR String)` beispielweise nicht erst eine Instanz der Klasse `Partner` geschaffen werden, sondern kann direkt mittels `Partner.findByPNR("A4711")` geschehen.

Kommentare können an jedes Element gehangen und mit beliebigem Freitext versehen werden. Im Beispiel wird ein Kommentar dazu benutzt eine Bedingung *(engl.Constraint)* an die Klasse `Partner` zu formulieren. *Constraints* sind besondere Bedingungen, die im Sinne einer Invariante an die Eigenschaften eines Modellelementes gestellt werden und vom modellierten System selbst kontrolliert sowie bei Verletzung adäquat und automatisch behandelt werden müssen. Wie genau diese Bedingungen etwa zur Laufzeit, im Beispiel durch ein Java Programm, forciert werden ist völlig freigestellt.
Stereotype und *Eigenschaftswerte* (auch *Tagged Values*) wurden in Kapitel 3 eingeführt und erläutert. Neu in die UML aufgenommene

Stereotype, die ein eigenes Klassensymbol besitzen, sind beispielsweise *Entity*, *Boundary* und *Control*.

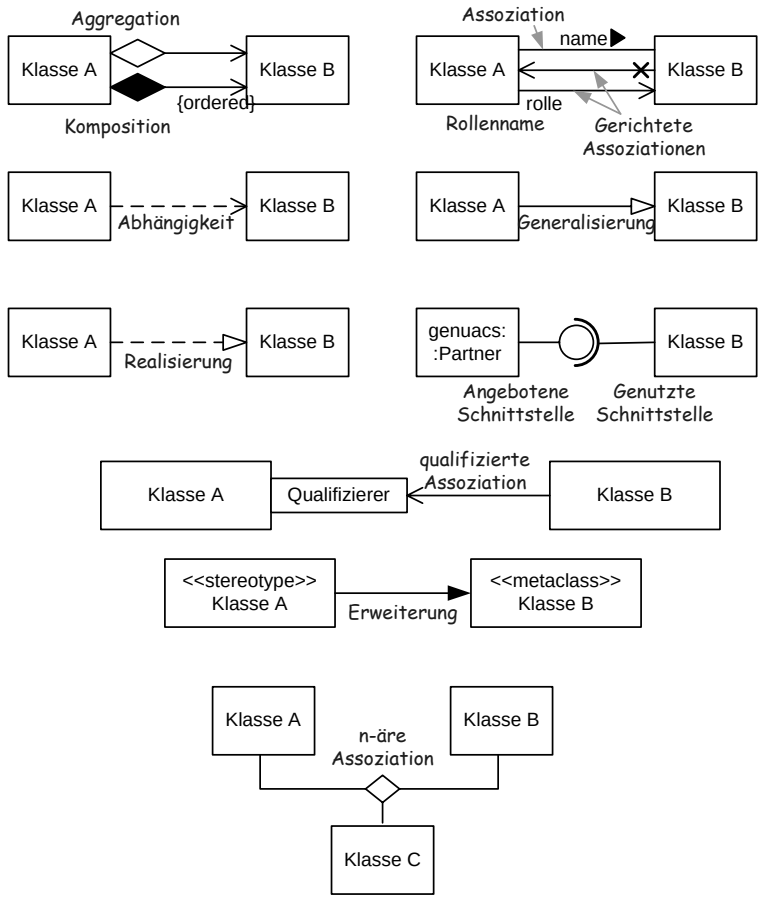


Abb. A.3 Beziehungen zwischen Klassen

Beziehungen zwischen Klassen

Beziehungen zwischen Klassen können in der UML durch die Verbindung der in Beziehung stehenden Klassen mittels Linien explizit dargestellt werden. Zur Übersicht ist der betreffende Ausschnitt aus der Gesamtübersicht noch mal in Abb. A.3 dargestellt. Die einfachste Art, mit der zwei Klassen in Beziehung stehen können, ist die der *Assoziation*. Sie wird dargestellt durch eine durchgehende Linie und bedeutet umgangssprachlich soviel wie „Klasse `A` und `B` kennen einander“. Fachlicher ausgedrückt spricht man davon, dass von einer Klasse zur anderen „navigiert“ werden kann. Sind Beziehungen auf eine Richtung beschränkt kann dies durch eine Pfeilspitze in die erlaubte Richtung dargestellt werden. Man spricht in diesen Fällen von *gerichteten Assoziationen*. Durch die Angabe eines Pfeils ist die *Na-*

vigation in die Gegenrichtung allerdings noch nicht explizit verboten. Will man diesen Sachverhalt darstellen, kann das Ende zu dem nicht navigiert werden darf symbolisch durchgestrichen werden.

Multiplizitäten von Assoziationen

Die Angabe der *Multiplizität* der Beziehung – also wie viele `Bs` der Klasse `A` bekannt sind – kann direkt an das betroffene *Assoziationsende* geschrieben werden. Standard ist die Annahme einer 1-zu-1 Beziehung. Diese Annahmen kann jedoch beliebig durch die Angabe eines oder mehrerer *Intervalle* geändert werden. So bedeutet die Angabe einer Multiplizität von `1..2, 3..6`, dass entweder ein bis zwei oder 3 bis 6 Elemente am betroffenen Assoziationsende vorhanden sein müssen. Die Eckpunkte der Multiplizitäten, im Beispiel also 1,2,3 und 6, heißen *Kardinalitäten*. Eine Sonderrolle spielt die Kardinalität `*`, die als *Joker (engl. Wildcard)* fungiert. Wird `*` angegeben bedeutet dies soviel wie „beliebig viele oder keine“. Das Intervall `1..*` wird gelesen als *1-bis-n*.

Durch die Vergabe von *Rollen* wird die Art der Beziehung einer Klasse zu einer anderen näher spezifiziert. Im obigen Beispiel existiert eine gerichtete Assoziation zwischen `Partner` und `Adresse`. Alle Elemente innerhalb `Partner` können unter dem Rollennamen `adressen` auf die Menge der dem aktuellen Partner zugeordneten Adressen zugreifen.

Teil-Ganzes Beziehungen (Aggregationen) werden durch die Dekoration des *Aggregats* (des „Ganzen“) mit einem Diamanten gekennzeichnet. Wird der Diamant zudem schwarz ausgefüllt, handelt es sich um eine *Komposition*. Folge einer Komposition ist die Abhängigkeit der Existenz der Teile vom Ganzen. Wird das Aggregat gelöscht, endet auch der Lebenszyklus der abhängigen Teile. Technisch spricht man in diesem Falle von *Kaskadiertem Löschen (engl. cascaded delete)*.

Vererbungen (Generalisierungen) werden mittels durchgehender Linie mit dicker Pfeilspitze modelliert. Gelesen wird in Pfeilrichtung „Klasse `A` erbt von Klasse `B`“ oder „`A` ist eine Art `B`“.

Realisiert eine Klasse die Funktionalität einer *Schnittstelle (engl. Interface)* oder benutzt sie einen durch eine Schnittstelle spezifizierten Service, kann dies durch die *Lollipop-Notation* angezeigt werden. Nach dem *Stecker-Prinzip* ist der runde Lollipop so zu interpretieren, dass ein Service angeboten wird, während das zugehörige konkave anzeigt, dass ein Service zur korrekten Funktionsweise in Anspruch genommen wird. Um anzuzeigen, dass eine Klasse ein Interface realisiert, kann alternativ das Interface als Klasse dargestellt

werden, die mit dem Schlüsselwort «interface» markiert ist. Soll dargestellt werden, dass eine Klasse die innerhalb eines Interface beschriebene Funktionalität zur Verfügung stellt, kann ein gestrichelter Pfeil vom realisierenden Element hin zur Interface-Klasse gezogen werden (*Realisieren-Beziehung*). Die Darstellung des gegenteiligen Sachverhaltes „Interface benutzen“ geschieht durch die Verwendung der *Abhängigkeitsbeziehung (engl.dependency)*. Mit ihr kann generell die Abhängigkeit eines Elementes von einem Anderen angezeigt werden. Da in unserem Falle eine Klasse von einer Funktionalität abhängig ist, die durch ein Interface realisiert wird, kann die Abhängigkeitsbeziehung die Benutzung des Interfaces modellieren. Notationsmittel für Abhängigkeiten ist der gestrichelte Pfeil mit offener Pfeilspitze modelliert.

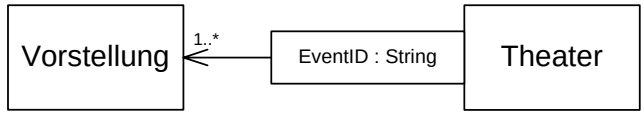


Abb. A.4 Beispiel für eine qualifizierte Assoziation

Qualifizierte Assoziationen liefern die Möglichkeit direkt auf bestimmte Objekte des „n-Endes“ einer zu-n-Beziehungen zuzugreifen. Der „Schlüssel“ zur Bestimmung dieser Teilmenge wird in einer Box an das Ende der Assoziation notiert, die den Zugriff auf ein dezidiertes Element des n-Endes benötigt. Abb. A.4 zeigt ein simples Beispiel. So sind einem Theater `1 bis n` Vorstellungen zugeordnet. Wollen wir durch Angabe der `EventID` direkt auf eine bestimmte Vorstellung zugreifen kann das durch die Angabe des dargestellten Qualifiers geschehen, der genau die Vorstellung zurückliefert, die der angegebenen ID entspricht.

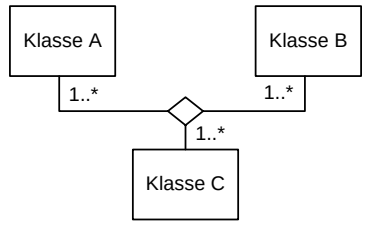
Auf die *Erweiterungsbeziehung (engl. extension)* wurde bereits in Kapitel 3 eingegangen. Sie spielt bei der Erweiterung bestehender Metamodelle mittels Profilen eine Rolle und soll hier nicht erneut erklärt werden.

N-äre Assoziationen bezeichnen die Möglichkeit Beziehungen zwischen mehr als zwei beteiligten Klassen anzugeben und sind eine der Neuerungen der UML in der Version 2. Notiert werden sie durch ein in der Mitte der Assoziationskanten platzierten hohlen Diamanten. Wichtige Einschränkung n-ärer Assoziationen im Vergleich zu den schon vorgestellten binären Versionen ist, dass sie keinerlei Angaben zur Navigierbarkeit enthalten dürfen. Grundsätzlich sind bei diesem Assoziationstyp alle Enden voll navigierbar. Die Semantik der n-ären Assoziation ist analog zu ihrem binären Gegenstück gehalten

und beschreibt eine beliebige Teilmenge der über den Diamanten verbundenen Klassen.

Als Beispiel folgt die Darstellung einer ternären Assoziation (Abb. A.5):

Abb. A.5 Ternäre Assoziation



Die Abbildung ist wie folgt zu interpretieren:

- Jede Instanz von Klasse `A` ist mindestens einer Kombination von Instanzen von `B` und `C` zugeordnet.
- Jede Instanz von Klasse `B` ist mindestens einer Kombination von Instanzen `A` und `C` zugeordnet und
- jede Instanz von Klasse `C` ist mindestens einer Kombination von Instanzen von `A` und `B` zugeordnet.

A.1.2 Objektdiagramm

Während Klassendiagramme das statische Metamodell eines Systems darstellen, dienen Objektdiagramme dazu eine Momentaufnahme eines Systems zur Laufzeit abzubilden und stellen eine mögliche Instanz ihres zugehörigen Klassenmodells dar. Abb. A.6 zeigt ein Beispiel, an dem die erlaubten Notationselemente und Konzepte erläutert werden.

Oben dargestellt das Klassendiagramm auf Metaebene M1 (vergleiche dazu Diskurs zu Metamodellierung in *Kapitel Modellierung*) und darunter dargestellt eine Situation, die zur Laufzeit des Systems auf M0 spielt und eine mögliche Instanz des M1 Modells darstellt.
Im Beispiel existiert eine Instanz `kunde` der Klasse `Partner`, die zum Zeitpunkt der Betrachtung die Werte `Cäsar`, `Little` und das Geburtsdatum `11/11/1911` besitzt. Diesem Objekt zugeordnet sind zwei Objekte des Typs `Kommunikation`, die mittels *Links* (Instanzen von Assoziationen) mit dem `kunde` Objekt verbunden sind.

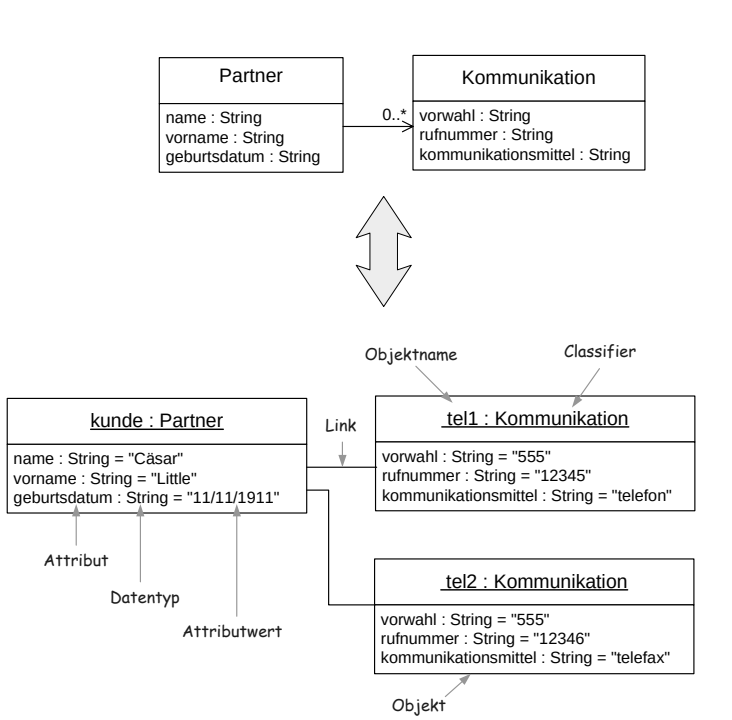
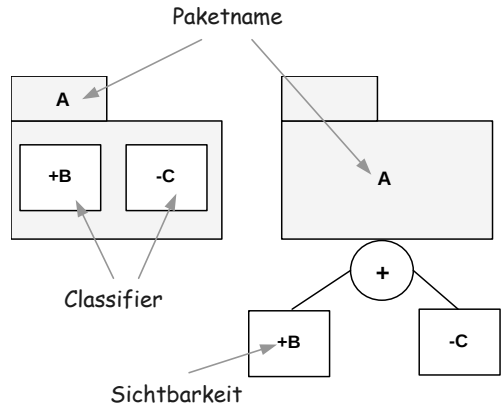


Abb. A.6 Übersicht Objektdiagramm

A.1.3 Paketdiagramm

Wie bereits im Kapitel über Modellierung (Kapitel 3) beschrieben, bildet die Strukturierung von Softwaresystemen in Einheiten (Module, Subsysteme, Schichten, …) ein grundlegendes Konzept zur Beherrschung von Komplexität. Einer der Mechanismen, die UML zu diesem Zweck bereitstellt, ist die Aufteilung von Systemen in *Pakete (engl. packages)*, in denen Elemente zu funktionalen Einheiten zusammengefasst und so die Sicht auf die im Paket enthaltenen Modellelemente zu einer Darstellung eines Namensraumes abstrahiert werden können. Abb. A.7 zeigt ein Beispiel für eine solche Strukturbildung.

Abb. A.7
Notation
Paketdiagramm



Pakete werden durch das Symbol eines stilisierten Ordners dargestellt. Der *Name* des Paketes erscheint wahlweise im *Reiter* (hier Paket `A`) oder im *Körper* des Paketes. Wird die erste Möglichkeit gewählt, werden die Elemente innerhalb des Paketes dargestellt (links), im zweiten Fall kann die rechte Darstellung verwendet werden.

Elemente eines Paketes können alle Konstrukte sein, die den Typ `PackagableElement` spezialisieren. Üblich sind die Elemente `Class` (wie hier durch die Klassen `B` und `C` dargestellt) oder Pakete selbst.

Pakete bilden einen eigenen *Namensraum*. Will man ein Element unter Angabe der enthaltenden Pakete adressieren (man spricht in diesem Falle vom *qualifizierten Namen* des Elementes, ansonsten vom *unqualifizierten*), stellt man einfach den Paketnamen (bzw. die Paketnamen, falls mehrere Pakete ineinander geschachtelt sind) vor den Namen des entsprechenden `PackagableElements` und trennt die einzelnen Bestandteile des so entstandenen qualifizierten Namens durch die Separatoren „`::`“ voneinander ab. In unserem Beispiel entspräche der qualifizierte Name von `C` somit `A::C`, eine Klasse, die in Java unter `java.util.List` deklariert ist, würde im entsprechenden UML-Diagramm unter `java::util::List` zu finden sein, usw.

Für die Sichtbarkeit der so geschaffenen Referenzen und die damit verbundenen *Modifier* gelten dieselben Regeln, die im Abschnitt über Klassendiagramme erläutert wurden.

Pakete können Elemente anderer Pakete *importieren (engl. import)*, *benutzen (engl. access)* oder ihre Elemente mit denen des anderer

Pakete *verschmelzen (engl. merge)*. Die graphische Darstellung dieser Sachverhalte ist in Abb. A.8 gegeben:

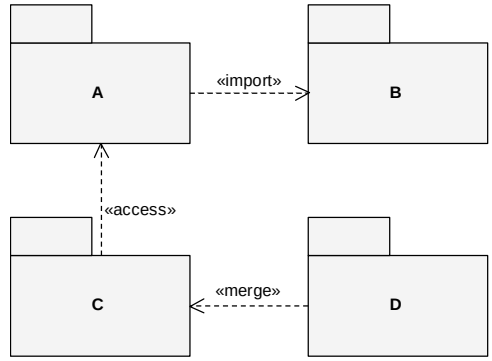


Abb. A.8 Beziehungen zwischen Paketen

Die Stereotype der Assoziationen bedeuten dabei Folgendes:

- `import`: Elemente aus importierten Paketen können unqualifiziert innerhalb des importierenden Paketes, verwendet werden. Zudem ist der Zugriff von außen auf die importierten Elemente durch eine Referenzierung über das importierende Paket möglich (z. B. `A`::B::x).
- `access`: Wie `import`, Unterschied ist jedoch, dass die importierten Elemente innerhalb des importierenden Paketes die Sichtbarkeit `private` besitzen, also nicht an andere Elemente außerhalb des Paketes weitergegeben werden können.
- `merge`: Durch die Anwendung dieser Assoziation entstehen im Quellpaket neue Elemente, die aus den ursprünglich im Quellpaket vorhandenen Elementen bestehen, die von den Elementen gleichen Namens des Zielpakets (das Paket auf das der Pfeil zeigt) erben. Für Details dieses, in der UML 2 vielgenutzten Mechanismus verweisen wir auf [UML2IS] sowie [JRH+03].

A.1.4 Komponentendiagramm

Komponentendiagramme (engl. component diagram) stellen das betrachtete System als klar durch Schnittstellen abgegrenzte Module dar. Die Betrachtung dieser Komponenten erstreckt sich im Laufe des Entwicklungszyklus von mehr organisatorischen Aspekten – Organisation in modulare Einheiten, bereitgestellte/verwendete Schnittstellen – hin zu Darstellungen, die sich bereits auf Build-/Deployvorgänge und die damit verbundene Verteilung des Systems zur Laufzeit beziehen.

In der MDA wird dieser Diagrammtyp vor allem zur Darstellung von Abhängigkeiten zwischen binären Komponenten sowie zur Visualisierung von Systemkontexten verwendet. Komponentendiagramme stellen ein wertvolles Mittel zur Modellierung des Lebenszyklus eines Systems dar, da durch diese der Werdegang eines Moduls oder Gesamtsystems von der statischen Quelle (z. B. Klasse) bis hin zur deployfähigen Binärkomponente (z. B. Artefakt) dargestellt werden kann.

Abb. A.9 Notation Komponentendiagramm

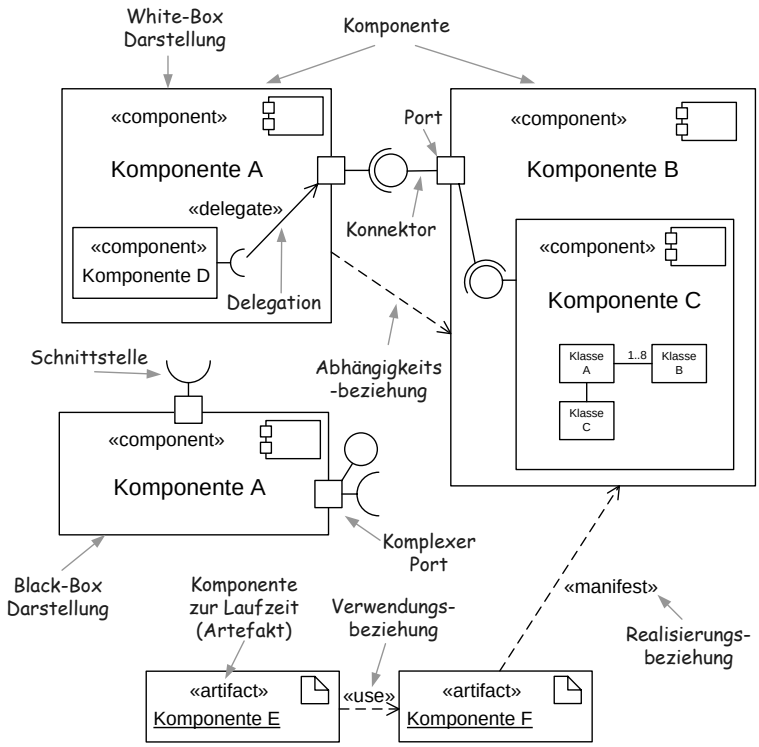


Abb. A.9 zeigt die Notationselemente des Komponentendiagramms. Komponenten erweitern die Metaklasse `Class` und werden entweder durch die Verwendung des Stereotypen «`component`» oder durch die Verwendung des Komponentensymbols in der rechten oberen Ecke des Rechtecks gekennzeichnet. Innerhalb von Komponenten wiederum können alle Elemente enthalten sein, die `PackageableElement` spezialisieren. Diese inneren Elemente realisieren, alleine oder in Zusammenarbeit, die Services, die die Komponente über ihre Schnittstellen anbietet. Über diesen Mechanismus können sie auch weitere Komponenten enthalten was eine beliebig tiefe Schachtelung erlaubt.

Zur Modellierung von Komponenten kann zwischen *White-Box* bzw. *Black-Box* Darstellungen gewählt werden. Wird die White-Box Darstellung verwendet, können die Beziehungen zwischen Ports und den unterliegenden inneren Strukturen des `Classifiers`, die die *Services* eines Ports realisieren dargestellt werden.

Obwohl nicht auf die Verwendung innerhalb des Komponentendiagramms beschränkt, bilden *Ports* ein wichtiges Modellierungselement dieses Diagrammtyps. Ports definieren die Punkte, an denen Interaktionen zwischen der Umgebung von `Classifiern` und den in ihnen enthaltenen Elementen – ihrer *internen Struktur* – stattfinden können. So wird eine klare Kapselung erreicht, die eine Entkopplung seines internen Aufbaus von dem ihn umgebenden Kontexte zur Folge hat und eine Wiederverwendung des `Classifiers` in jeder Umgebung ermöglicht, die die an den Port geknüpften Bedingungen erfüllt.

So ermöglichen Ports die Verknüpfung von Schnittstellen eines `Classifiers` mit denjenigen internen Elementen, die den angebotenen Service der Schnittstelle realisieren. Man spricht in diesem Falle davon, dass eine Schnittstelle die an ihr zur Verfügung gestellte Funktionalität an die entsprechenden Elemente im inneren des `Classifiers` *delegiert*. Im Beispiel benötigt `Komponente D`, die in `Komponente A` enthalten ist eine Funktionalität, die an einen Port von `A` delegiert wird. `Komponente B` wiederum stellt diese Schnittstelle zur Verfügung, realisiert die Funktionalität jedoch auch wiederum nicht selbst, sondern lässt diese Aufgabe `Komponente C` übernehmen. Delegationen werden durch eine gerichtete Assoziation mit Stereotyp `«delegate»` dargestellt und können Zwischen angebotenen und verwendenden Ports angetragen werden. Sie stellen so einen Vertrag zwischen Port und einem Verhalten (`Behaviour`) dar, das die Anfrage des Ports realisiert.

Nicht nur angebotene, sondern auch benötigte Interfaces können an einen Port angetragen werden. In diesem Falle kann mit demselben Mechanismus genau spezifiziert werden, welche Elemente des `Classifiers` die Services konsumieren.

Das Symbol eines Ports ist ein kleines Rechteck am Rand des `Classifiers`. Wird das Symbol innerhalb der Umrandung modelliert, bedeutet dies, dass der Port für Elemente außerhalb der Umrandung nicht sichtbar ist. Schnittstellen, die mit einem Port verbunden sind – und einen wesentlichen Teil der an dessen Benutzung geknüpften Bedingungen darstellen – werden mit der bereits im Abschnitt über Klassendiagramme erwähnten *Lollipop-Notation* dargestellt (vgl. Abschnitt A.1.1). Ist mehr als eine Schnittstelle mit einem Port verbunden, spricht man von *komplexen Ports*.

Artefakte

Als letztes Element wollen wir noch die *Artefakte* betrachten. Sie stellen im Allgemeinen alle Arten von physischen Einheiten zur Laufzeit des Systems wie Files, ausführbare Dateien, Modelle usw. dar. Im Komponentenmodell repräsentieren sie binäre Instanzen einer Komponente zur Laufzeit und stehen damit zu ihnen in etwa in der Beziehung, wie Objekte zu Klassen. Der Sachverhalt, dass ein Artefakt eine Instanz einer Komponenten ist, wird durch die Abhängigkeitsbeziehung mit Stereotyp «manifest» dargestellt. Achtung hier findet sich eine Änderung im Vergleich mit der UML 1.x, in der diese Beziehung durch das Schlüsselwort «implement» ausgedrückt worden ist. «use» wiederum beschreibt die *Verwendungsbeziehung* zwischen Artefakten (vgl. Abb. A.9).

A.1.5 Verteilungsdiagramm

Das *Verteilungsdiagramm (engl. deployment diagram)* beschreibt zum einen die technische Infrastruktur eines Systems, zum anderen jedoch auch die physische Verteilung der binären Software-Komponenten auf dieser Hardware. Abb. A.10 zeigt die Notationselemente, die die UML zur Modellierung dieser Aspekte bereitstellt.

Hauptnotationselement ist der *Knoten (engl. node)*, der eine Einheit repräsentiert, die in irgendeiner Weise zur Ausführung eines Artefaktes zur Laufzeit beiträgt. Die Notation für einen Knoten ist ein perspektivischer Quader, der den Namen und den Typ des Knotens enthält. Die Darstellung eines Knotens ist ähnlich der Notation von *Objekten* (vgl. Abschnitt A.1.2) und wird hier nicht noch einmal beschrieben.

Knoten können *Geräte (engl. devices)* oder *Ausführungsumgebungen (engl. execution environmens)* darstellen und werden dann entsprechend mit dem Stereotypen «device» bzw. «execution environment» gekennzeichnet. Geräte stellen Ressourcen mit Rechenkapazität dar, während Ausführungsumgebungen auf Knoten installierte Plattformen darstellen und beliebig ineinander geschachtelt werden können.

Kommunizieren Geräte miteinander kann dies durch eine *Assoziation* dargestellt werden. Wie auch bei Klassen können diese Assoziationen gerichtet oder ungerichtet sein. Artefakte (vgl. Abschnitt A.1.4) stellen binäre Instanzen von Komponenten dar und können auf Geräten installiert, verteilt bzw. deployed werden. Die Beziehung, die diesen Sachverhalt ausdrückt, ist die *Deployment Beziehung*. Sie wird durch eine Abhängigkeit mit Stereotyp «deploy» dargestellt. Zur Detaillierung dieser Beziehung kann eine zusätzliche

Spezifikation dieses Deployments an das oder die betroffenen Artefakte gehangen werden. Dargestellt als Objekt können hier Sachverhalte modelliert werden, die die Art der Verteilung näher erläutern, und damit dem Konzept des *Deployment-Deskriptors* aus dem Java EE Umfeld entsprechen.

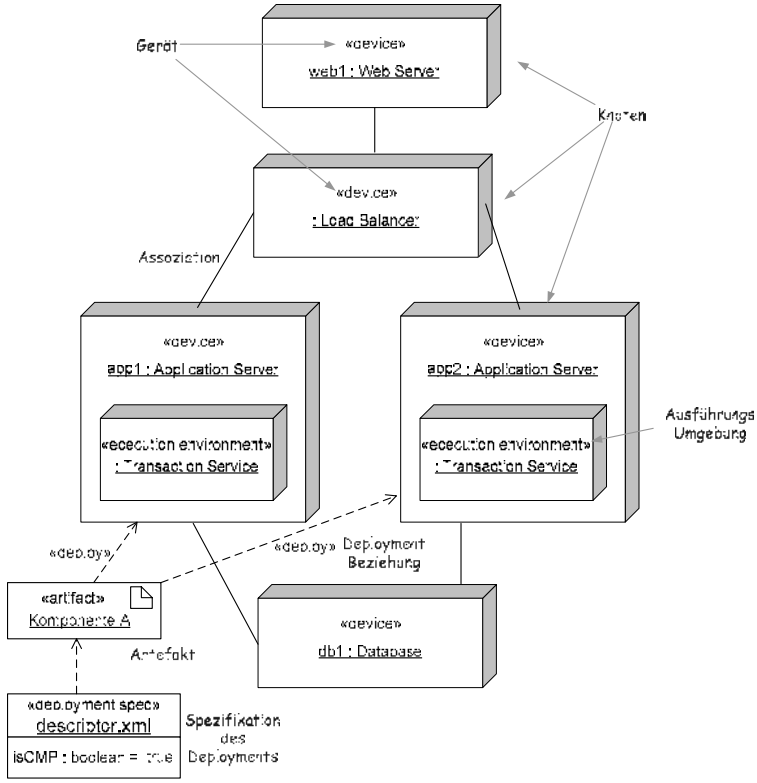


Abb. A.10 Notation Verteilungsdiagramm

A.1.6 Kompositionsstrukturdiagramm

Die *Kompositionsstruktur (engl. composite structure)* beschreibt ein System als eine Menge verbundener Instanzen, die zur Laufzeit zusammenarbeiten, um eine bestimmte Aufgabe zu erledigen. Die Zusammenarbeit zeigt sich gewöhnlich in einer Kommunikation wie etwa dem Austausch von Signalen, Aufrufe von Operationen usw. und wird visuell dargestellt durch eine *Konnektor* genannte Assoziation. Weiterhin dargestellt wird der innere Aufbau der Instanzen – ihre *Teile (engl. parts)* – sowie die Interaktionspunkte (Ports, vgl. auch Abschnitt A.1.1) zwischen denen die Kommunikation der Instanzen stattfindet.

Die an einer Komposition teilnehmenden Elemente – in UML-Slang *Entitäten (engl. entities)* genannt – können so unterschiedlicher Art sein wie Use Cases, Komponenten, Klassen,Objekte, Attribute, Operationen usw. Auf den Typ bezogen bedeutet dies, das alle `ConnectableElements` zur Darstellung im Diagramm in Frage kommen. Neben den im folgenden Beispiel gezeigten und bereits erwähnten Konnektoren können auch alle weiteren typspezifischen Beziehungsdarstellungen zwischen den dargestellten Elementen verwendet werden. Dies sind im Falle von Klassen z. B. die Generalisierung und die Aggregation, im Falle von Komponenten die Delegation, usw.

Ergebnis der Zusammenarbeit ist ein *Verhalten (engl. behavior)* des Systems. Die nähere Identifizierung der Aufgaben der einzelnen Teile zur Realisierung dieses Verhaltens geschieht durch die Zuordnung von *Rollen*. Um die Komplexität der Systembeschreibung zu verringern werden pro Kollaboration nur diejenigen Rollen und ihr (Teil am) Verhalten beschrieben, die zur Beschreibung eines klar abgegrenzten *Aspekts* des System benötigt werden.

Abb. A.11 Kompositionsstruktur – Klassendarstellung

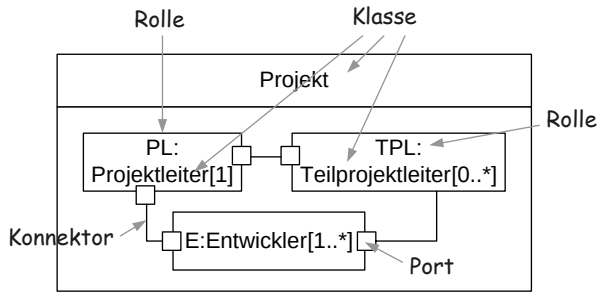


Abb. A.11 zeigt die Darstellung einer Dekomposition der Klasse `Projekt` sowie den kommunikativen Zusammenhang ihrer internen Struktur. Ein `Projekt` besteht aus genau einem `Projektleiter` eventuellen `Teilprojektleitern` (optional) sowie mindestens einem `Entwickler`. Die zugehörigen Rollen sind entsprechend `PL`, `TPL` und `E`.

Neben der Klassendarstellung gehört auch die *Komponentendarstellung* zu den strukturgetriebenen Kompositionsmodellen. Diese wurde bereits in A.1.4 ausgiebig beschrieben.

Kollaborationen

Mit Hilfe einer *Kollaboration (engl. collaboration)* kann die *Zusammenarbeit* der Modellelemente auf einer mehr verhaltensbezogenen Ebene beschrieben werden.

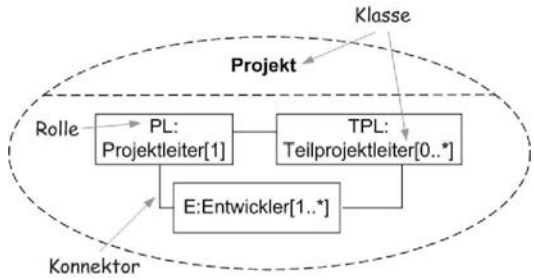


Abb. A.12 Kollaboration – Notation 1

Abb. A.12 zeigt das eben vorgestellte Beispiel in der alternativen Darstellungsweise der Kollaborationsnotation. Abb. A.13 zeigt eine weitere Alternative.

Hier werden die spezifischen Darstellungsmöglichkeiten der Klassenelemente benutzt und die Datenstrukturen der beteiligten Elemente, die zur Erfüllung der jeweiligen Aufgaben erforderlich sind angeführt. Erbrachte Services der Klassen könnten auf diese

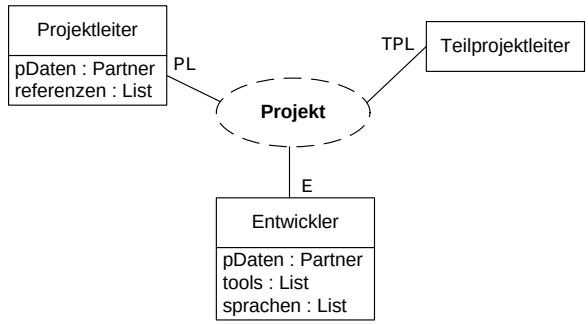


Abb. A.13 Kollaboration – Notation 2

Weise zum Beispiel als Operationen angetragen werden.

Einsatz von Kollaborationsdiagrammen

Kollaborationen beschreiben die Rollen, die zur Erfüllung einer genau abgegrenzten Aufgabe benötigt werden. Die tatsächliche Erfüllung der Aufgabe geschieht, indem Elemente, die die in Form dieser Rollen beschriebenen Strukturen und Eigenschaften besitzen, in der konkreten Situation instanziiert werden und ihr Verhalten zur Ausführung gelangt. Ihrem Wesen nach eignen sich Kollaborationen daher vor allem zur Beschreibung allgemein gültiger Muster zur Lösung eines bestimmten Problems und werden häufig zur Dokumentation von *(Design-)Patterns* genutzt.

A.2 Verhaltensdiagramme

Neben den *Strukturdiagrammen*, die wir in der ersten Hälfte dieses Kapitels unter die Lupe genommen haben, bilden die *Verhaltensdiagramme* die zweite große Gruppe von Diagrammen der UML. Sie *spezifizieren* das Verhalten eines Systems. *Verhalten (engl. behavior)* selbst entsteht durch die Ausführung von *Aktionen (engl. actions)* durch Objekte und manifestiert sich im Wechsel der nach außen sichtbaren oder inneren Zustände des Gesamtsystems. Die Art dieser Zustände sowie die Wertebereiche der möglichen Zustände werden durch die Struktur des Systems beschrieben und waren Thema im ersten Teils dieses Anhangs.

Einordnung ins UML Metamodell

Im Bezug auf das unterliegende Metamodell erben alle *Verhaltensbeschreibungen* von der Klasse `Behavior`. `Behaviors` können `Classifiern` zugeordnet sein und beschreiben (spezifizieren) dann dessen Verhalten. Man nennt man den `Classifier`, dem das Verhalten zugeordnet ist – die Zuordnung geschieht mittels Kompositionsbeziehung – den *Kontext* in dem das Verhalten stattfindet. Der Kontext eines Verhaltens bestimmt so unter anderem auf welche Elemente während der Ausführung zugegriffen werden kann.

Möglichkeiten zur Verhaltensbeschreibung

Die Spezifizierung von Verhalten geschieht in der UML also mit Elementen des Typs `Behavior`. Direkt von `Behavior` erben die folgenden Klassen, die in eigenen Modellarten dargestellt und so direkt zur Verhaltensmodellierung verwendet werden können:

- `StateMachine` (s. *Zustandsautomat* Abschnitt A.2.3), beschreibt Ereignisse, die Zustandsänderungen in Objekten verursachen und weitere Verhaltensaufrufe durchführen,
- `Activity` (s. *Aktivitätsdiagramm* Abschnitt A.2.2), hier wird der Fokus auf die Reihenfolge, Bedingungen, sowie den Aufruf weiterer Verhalten und deren Ein-/ und Ausgaben gelegt,
- `UseCase` (s. *Use Case Diagramm* Abschnitt A.2.1), eine Möglichkeit der Darstellung von Verhalten aus der Sicht von Akteuren außerhalb des Systems, um eine Black-Box Sicht auf das System zu erhalten, interessant in diesem Kontext: Use Cases können selbst durch die Zuordnung von Verhalten näher spezifiziert werden.
- Sowie `Interaction` und damit alle *Interaktionen bzw. die zugehörigen Diagramme* (siehe dazu Abschnitt A.2.4), die den Aufruf von Verhalten als Ursache des Austausches von Nachrichten zwischen Objekten modellieren.

Ein Systemverhalten kann durch alle oben aufgeführten oder selbst definierte neue Verhaltensdiagramme spezifiziert werden. Die Auswahl welche Diagrammart zur Darstellung am besten geeignet ist geschieht durch Erfahrung und die Bewertung welche darzustellenden Aspekte am einfachsten und verständlichsten umgesetzt werden können.

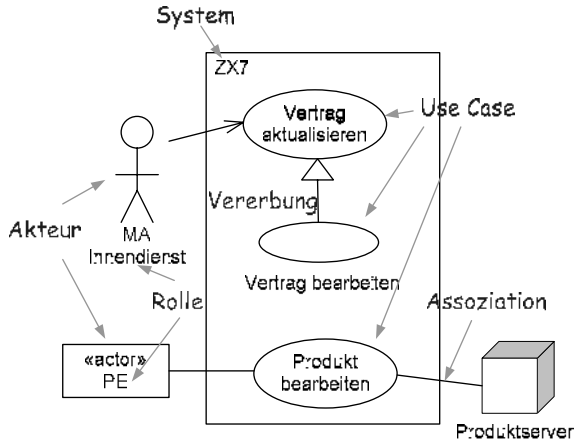
A.2.1 Use-Case Diagramm

Das *Use-Case Diagram (dt. Anwendungsfalldiagramm)* stellt die Beziehungen zwischen *Akteuren*, *Anwendungsfällen (engl. use cases)* und dem modellierten *System* dar. Akteure sind definiert als die Menge aller Entitäten, die mit dem System interagieren. Akteure können somit sowohl weitere Systeme, als auch menschliche Benutzer darstellen. Gemein ist beiden Arten von Akteuren, dass es sich immer um vom betrachteten System aus externe Entitäten handelt. Interagieren zwei Entitäten auf die selbe Art und Weise mit dem System, sind beide Ausprägungen der gleichen Akteursrolle. Akteure stellen also keine Instanzen dar (ein bestimmter Benutzer „Karl“ oder einen bestimmtes System „HAL9000“), sondern sind vielmehr als eine Art „Typ“ (Rolle) aufzufassen. Sie beschreiben das Verhalten von Entitätsklassen in einem bestimmten Kontext (vergl. *Kollaborationen* in Abschnitt A.1.6). Entitäten der realen Welt können in verschiedenen Kontexten mehrere Rollen besitzen („HAL9000“ ist *Host* von Software und mit dem betrachteten System kooperierendes *Lohnabrechnungssystem*).

Anwendungsfälle (Use Cases) stellen entweder ein Ziel dar, das ein System erfüllen, oder ein Ergebnis, das es liefern muss um den Anforderungen, die in der Spezifikation an es gestellt wurden zu genügen. Ähnlich wie das Element *Akteur* liefern Anwendungsfälle nur die allgemeine Beschreibung von *möglichen* Verhalten, die in einer konkreten Situation (Belegung von Objekten mit Datensätzen, Festlegung von Zeitpunkten, …) instanziiert werden muss, um ausgeführt und so beispielsweise getestet werden zu können. Man spricht bei der Instanz eines Anwendungsfalles von einem *Szenario*.

Abb. A.14 zeigt die Notationselemente des Anwendungsfalldiagramms.

Abb. A.14 Notation Use Case Diagramm



Der *Systemkontext* wird durch ein Rechteck dargestellt, dessen Umrandung die Grenzen des betrachteten Systems darstellt. Die Darstellung von *Akteuren* geschieht durch das bekannte Strichmännchen oder durch die Anwendung des Stereotypen `«actor»`. Alternativ kann einer Rolle auch ein eigenes Symbol zugeordnet werden. Hier hat man die freie Auswahl. Weit verbreitet sind die Box zur Darstellung eines Systems sowie die Uhr zur Modellierung eines zeitgesteuerten Zugriffs.Die Zuordnung von Akteuren zu Anwendungsfällen geschieht durch die Anwendung von *Assoziationen*. Sind diese gerichtet, bedeutet dies, das der zugehörige Akteur die Ausführung des Use Cases anstößt. Bei einer sauberen Rollendefinition stößt immer nur ein Akteur den Use Case aktiv an, alle anderen assoziierten Rollen sind lediglich in das abgebildete Verhalten eingebunden.

Beziehungen zwischen Use Cases

Die *Generalisierung* ist eine der erlaubten Beziehungen von Use Cases untereinander. Die Bedeutung entspricht der Generalisierung von Klassen. Erbt ein Anwendungsfall von einem anderen, besitzt er alle dort aufgeführten Schritte, kann diese jedoch überschreiben oder spezialisieren bzw. verfeinern. In der Abbildung nicht aufgeführt, jedoch auch vorgesehen ist die Vererbungsbeziehung zwischen Akteuren. Ein spezialisierender Akteur erbt alle Beziehungen zu Anwendungsfällen von seinem Elternteil und kann zusätzliche hinzufügen.

Weitere erlaubte Beziehungselemente von Use Cases untereinander sind in Abb. A.15 dargestellt:

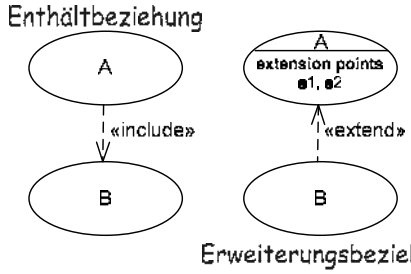


Abb. A.15 Beziehungen zwischen Use Cases

Die *Enthältbeziehung (engl. include)* zeigt an, dass im Verlaufe des Anwendungsfalls `A` immer auch die Schritte ausgeführt werden, die in Anwendungsfall `B` beschrieben sind. Damit entspricht die Beziehung so etwas wie einem Aufruf von `B` aus `A` an einer fest definierten Stelle. Dieser Aufruf ist weiterhin nicht an Bedingungen geknüpft, sondern findet immer statt. Wo er stattfindet ist im erweiterten Use Case beschrieben (im Beispiel UC `A`). UC `B` „weiß" nichts von dieser Wiederverwendung, was die Wiederverwendung in mehr als einem Kontext ermöglicht.

Die *Erweiterungsbeziehung (engl. extension)* beschreibt eine oder mehrere mögliche Erweiterungen von Use Case `A` an fest definierten *Erweiterungspunkten (engl. extension points).* An ihnen kann Funktionalität, die in anderen Anwendungsfällen beschrieben ist, „eingehängt" werden. In Abb. A.15 sind dies die Erweiterungspunkte `e1` und `e2`. Im Gegensatz zur Enthältbeziehung ist sich der erweiterte Use Case (im Beispiel UC `A`) der Erweiterung nur in so fern bewusst, dass in ihm die Erweiterungspunkte definiert werden. Wann, ob und von wem eine solche Erweiterung stattfindet entscheidet der erweiternde Anwendungsfall (hier UC `B`). Dies macht UC `A` anpass- und in mehreren Kontexten wiederverwendbar, während UC `B` auf die Verwendung zusammen mit UC `A` festgelegt ist.

A.2.2 Aktivitätsdiagramm

An dieser Stelle wollen wir uns das *Aktivitätsdiagramm* näher anschauen. Das Spektrum der zur Verfügung stehenden Abstraktionsniveaus, die mit dieser Diagrammart modelliert werden können, reicht von der Beschreibung des konkreten Verhaltens einer Operation innerhalb einer Klasse, bis hin zur Modellierung ganzer Geschäftsprozesse. Aktivitätsdiagramme stellen so etwas wie den „Allrounder" der Verhaltensmodellierung dar, weshalb wir Ihnen an dieser Stelle besondere Aufmerksamkeit schenken wollen.

Die Spezifikation des Aktivitätsdiagramms hat beim Übergang zur Version 2 von allen Diagrammarten mit die meisten Änderungen er-

fahren. War das Aktivitätsdiagramm in vorherigen Versionen eher eine Sonderform des *Zustandsdiagrammes*, liegt ihm in der Version 2.x der UML ein völlig eigenständiges Konzept zu Grunde. Der Ablauf einer Aktivität, die aus einer Menge von Aktionen besteht, wird nun durch eine den Petri-Netzen ähnliche Semantik geregelt (*Token-Konzept*), was der Simulation und der formalen Analyse von Modellen eine völlig neue Grundlage bereitet. Zusätzliche Konstrukte zur Ablaufsteuerung (*for-Schleifen*, *if-Bedingungen*, etc.) stellen die in den 1.x Versionen zwar vorhandenen, aber umständlich zu realisierenden, Möglichkeiten zur Kontrollflussmodellierung bereit, sollten jedoch, wie übrigens viele Features der neuen UML-Version, sparsam eingesetzt werden, da die Komplexität der Diagramme sonst den Nutzen der visuellen Darstellung aufwiegt.

Bei der Anpassung des Aktivitätsdiagramms hat die OMG versucht eine bestmögliche Unterstützung des MDA-Ansatzes zu gewährleisten. Die Integration der Prozessmodellierung in den Entwicklungsprozess – und damit die Konservierung von teurem und hochspezialisiertem Fachwissen – sowie die daraus resultierende, automatische Generierung von Ablaufsteuerungen innerhalb von Softwaresystemen, ist eines der großen Ziele (und Herausforderungen!) innerhalb der modernen Softwareentwicklung, und die Wiederverwendung dieses Wissens, dass sich auf sehr abstraktem Niveau befindet, im Gegensatz zur oftmals problematischen Wiederverwendung feingranularer binärer Komponenten, scheint auf dem Wege der breiteren Verwendung dieser dynamischen UML-Diagramme, einen Schritt näher zu rücken.

Abb. A.16 zeigt ein Modell einer Beispielaktivität, das die gängigsten und für dieses Buch wichtigsten Notationselemente enthält. Auf Grund der Fülle von möglichen Notationen soll an dieser Stelle erstmals von der Vorgehensweise der vorigen Abschnitte abgewichen und nur ein Ausschnitt der möglichen Notationen anhand eines Beispiels erläutert werden. Zusätzlich werden in den Kapiteln in denen das Aktivitätsdiagramm intensiv genutzt wird noch nicht besprochene Elemente erklärt, so dass am Ende ein guter Überblick über die Verwendung dieses für die MDA wichtigen Werkzeugs beim Leser vorhanden sein sollte. Für einen vollständigen Überblick verweisen wir auf die am Schluss des Abschnittes aufgeführte Literatur, die neben Buchempfehlungen auch Verweise auf ausgezeichnete frei verfügbare Beschreibungen dieses UML Konstruktes enthält.

Ein *Aktivitätsdiagramm* beschreibt eine *Aktivität* als einen Ablauf von *Aktionen*. Dazu wird eine mögliche Abfolge – im Folgenden

auch *Kontrollfluss* genannt – von Knoten modelliert. Die Knoten repräsentieren *Aktionen*, die in einer Vielzahl spezieller Unterarten vorkommen und zum Teil auch eigene Notationen besitzen können. Neben den Aktionsknoten werden im Wesentlichen noch zwei weitere Knotenarten unterschieden: *Objektknoten*, die Daten repräsentieren und als Eingabe bzw. Ausgabe von Aktionen dienen sowie *Kontrollknoten* (Entscheidungen, …) die den Fluss des Verhaltens steuern.

Mögliche Pfade durch die Aktivität werden durch gerichtete Assoziationen dargestellt, die *Kanten* genannt werden. Eine Aktivität startet immer an ihrem *Startknoten* (ein ausgefüllter schwarzer Punkt) und endet mit dem Aufruf eines weiteren Verhaltens (z.B. einer Folgeaktivität), oder beim Erreichen eines *Endknotens*.

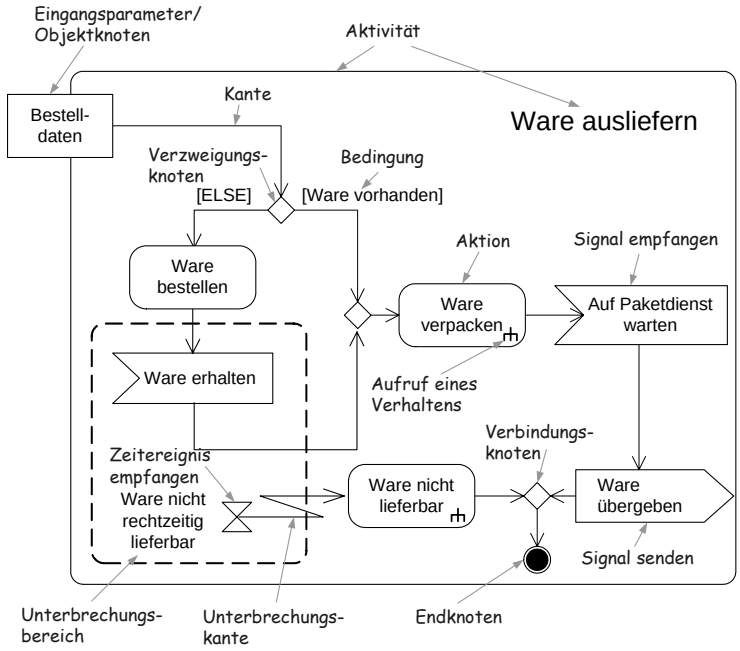


Abb. A.16 Notation Aktivitätsdiagramm

Im Beispiel ist eine Ausnahme von der Regel festgehalten. Die Aktivität `Ware ausliefern` startet sobald `Bestelldaten` am Eingang anliegen. Zur Vereinfachung der Darstellung kann in einem solchen Falle die explizite Angabe des Startknotens entfallen.

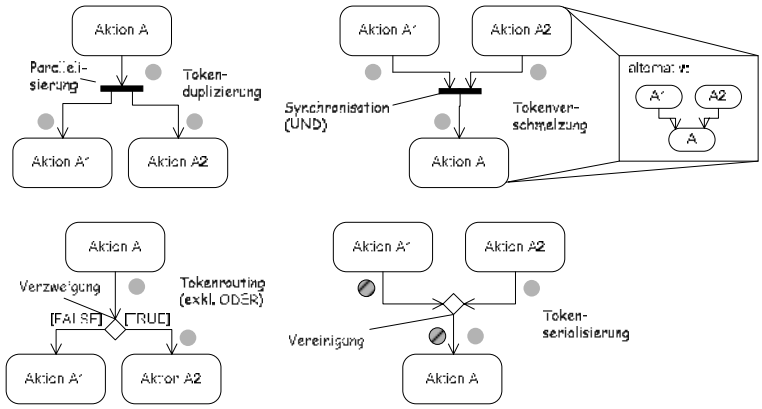
Kontrollflüsse

Der *Kontrollfluss* einer Aktivität wird durch ein *Token* versinnbildlicht, das an den Kanten „entlangrollt“. Der Weg durchs Diagramm, der tatsächlich in der konkreten Situation genommen wird entspricht so dem weg dieses Tokens und wird durch die bereits erwähnten

Kontrollknoten gesteuert, die den Weg des Tokens beeinflussen können. Wenn keine gegenteilige Angabe gemacht wird, werden eingehende Token dabei grundsätzlich in FIFO-Ordnung (*First-In-First-Out*) an die entsprechende ausgehende Kante weitergeleitet.Das Token-Konzept verdeutlicht Abb. A.17.

Bei der *Parallelisierung* (links oben dargestellt) werden eingehende Token vervielfältigt, so dass an jeder ausgehenden Kante wieder ein Token anliegt. Es entstehen so zusätzliche, gleichzeitige Flüsse innerhalb der Aktivität, die parallel zueinander ablaufen.

Abb. A.17 Token-Konzept



Sychronisation und Parallelisierung

Bei der *Synchronisation* (oben rechts) werden die an den Eingängen ankommenden Token zu einem einzigen Token verschmolzen. Sie stellt damit das Gegenstück der eben vorgestellten Parallelisierung dar. Die Verschmelzung findet erst dann statt, wenn an allen Eingängen ein Token anliegt. In Ausdrücken der Logik gesprochen realisiert die Synchronisation damit ein *UND-Konstrukt*, da die Ausführung des Verhaltens erst dann weitergeht, wenn an allen Eingängen des Synchronisationsknotens ein Token anliegt.

Verzweigung und Vereinigung

Die *Verzweigung (Entscheidungsknoten, engl. decision node)* (unten links) realisiert ein Routing von Token. An jeder ausgehenden Kante ist eine Bedingung angetragen, die in eckigen Klammern angegeben wird und einen zu `true` oder `false` auswertbaren Ausdruck darstellen muss. Um ein deterministisches Verhalten zu garantieren, müssen diese Bedingungen zwingend disjunkt sein und realisieren dann ein *XOR-Konstrukt*. Sind mehrere Bedingungen gleichzeitig erfüllt entscheidet der Zufall auf welche Reise das Token geschickt wird. Kann keine Kante schalten, da alle Bedingungen zu `false` ausgewertet werden, bleibt das Token hängen. Um stehts zumindest einen Ausweg zu garantieren, empfiehlt es sich stets eine

Kante mit der speziellen Bedingung `[else]` zu belegen, die genau dann `wahr` wird, wenn alle anderen Bedingungen `falsch` sind.

Die *Vereinigung (engl. merge)* entspricht einer Serialisierung von Token. Eingehende Token werden ohne weitere Auswertung, wiederum in FIFO-Ordnung, an die ausgehende Kante weitergereicht. Bitte beachten Sie, dass alle Alternativflüsse, die durch die Verwendung einer Verzweigung entstehen und wieder in einer gemeinsamen Aktivität enden oftmals explizit mittels der Vereinigung zusammengeführt werden müssen, um das gewünschte Verhalten zu modellieren. Werden mehrere Kanten direkt an eine Aktion angetragen entspricht dies einer *implizite Synchronisation*, also einem *UND*, was oftmals nicht dem eigentlich gewünschten Verhalten entspricht! Dargestellt ist diese Situation in Abb. A.17 oben rechts. Verlassen umgekehrt mehr als eine Kante eine Aktion direkt, ohne zwischengeschalteten Kontrollknoten hat dies die Bedeutung einer *impliziten Parallelisierung*. Ist Aktion `C` beendet legt sie somit je ein Token auf alle ausgehenden Kanten Auch hier ist also Vorsicht geboten, um tatsächlich das gewünschte Verhalten zu modellieren. Abb. A.18 stellt diese Zusammenhänge noch einmal grafisch dar:

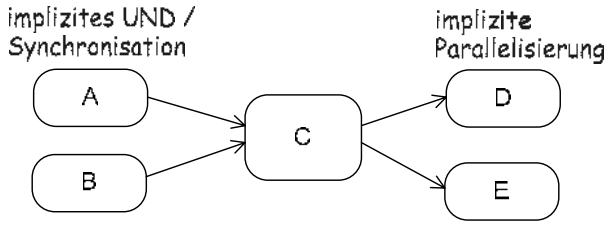
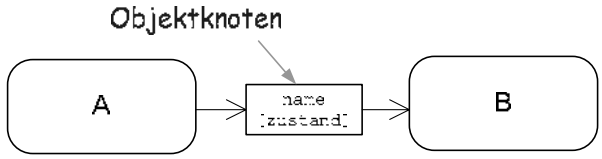


Abb. A.18 Implizites im Tokenfluss

Objektflüsse

Objektflüsse sind Flüsse, die auf ihren Kanten Datenpakete in Form von *Objektknoten* mit sich führen. Diese können Aktionen als *Parameter* dienen oder stellen das Ergebnis eines Verhaltens am Ausgang eines Knotens dar. Beachtet werden muss, dass adjazente Knoten nach dem Stecker-Prinzip kompatible Ein-/Ausgabe Parameter besitzen müssen. Objektflüsse besitzen auch steuernden Charakter. Besitzt eine Aktion einen Eingabeparameter, beginnt sie erst dann mit der Ausführung, wenn an ihrem Eingang ein entsprechendes Objekt anliegt. Zur Modellierung von Objektflüssen existieren verschiedene Darstellungsalternativen. Die einfachste Darstellung zeigt Abb. A.19.

Abb. A.19 Einfacher Objektfluss



Nachdem Aktion `A` beendet ist, verlässt der Objektknoten `name` als Ergebnis die Aktion und dient Aktion `B` als Eingabeparameter. Optional kann der Zustand des Objektes in eckigen Klammern angegeben werden. Zur Einbettung des Objektes in einen übergeordneten Kontext kann es notwendig sein die zugehörige Klasse auszumodellieren. Dies geschieht wie gewohnt in einem Klassendiagramm und ist hier nicht dargestellt.

Alternativ können Objektflüsse auch in der so genannten *Pin-Notation* angegeben werden (vgl. Abb. A.20). Die Alternativdarstellungen haben in allen Situationen die gleiche Bedeutung.

Parameter können in *Parametersätzen* geordnet werden, um Auswahlmöglichkeiten zu modellieren.

Abb. A.20 Parametersatz

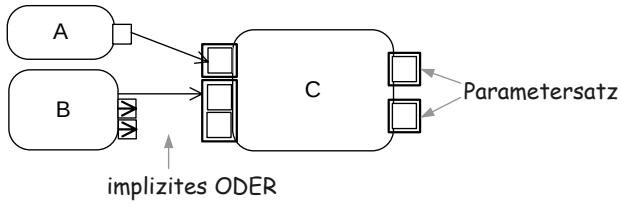


Abb. A.20 zeigt die Verwendung dieses Konstruktes. Aktion `C` besitzt zwei Sätze an Eingabeparametern, von denen mindestens einer bedient werden muss, damit die Aktion ausgeführt werden kann. `A` ist kompatibel zum ersten Satz, `B` entsprechend zum Zweiten. Die Ausführung von `C` beginnt, wenn an einem der beiden Sätze ein Datenpaket anliegt. Liegen an beiden Sätzen gleichzeitig Daten an, entscheidet das Los.

Feinheiten

Nochmals explizit erwähnt werden soll an dieser Stelle, dass das Tokenkonzept und damit auch die Beeinflussung von Objektflüssen durch Kontrollknoten in der Basis nicht von den zuvor besprochenen Kontrollflüssen unterscheidet, die beiden Konzepte im Detail jedoch Unterschiede aufweisen, die man kennen muss.

Lässt man bei mehreren ein- oder ausgehenden Objektflüssen in eine Aktion die Kontrollknoten weg, ist auch hier Vorsicht geboten,

da sich die unterliegenden Prinzipien teilweise vom intuitiv erwarteten Verhalten unterscheiden. Abb. A.21 zeigt diese Sonderfälle:

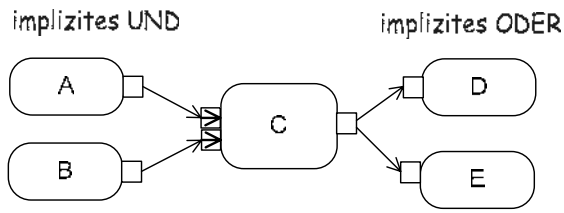


Abb. A.21 Implizites im Objektfluss

Eine Aktion kann erst starten, wenn an allen Dateneingängen kompatible Objekte anliegen, hier existiert also ein *implizites UND*. Hier ist die Interpretation also genauso wie bei den zuvor besprochenen Kontrollflüssen.

Anders als ein Token bei Kontrollflüssen jedoch kann sich ein *ausgehendes* Datenpaket einer Aktion nicht ohne explizite Parallelisierung aufteilen. Stehen also prinzipiell mehr als ein Abnehmer eines ausgehenden Objektes zur Verfügung, entscheidet das Los welche Folgeaktion das Paket bekommt. Hier liegt also ein *implizites (Exklusiv-)ODER* vor. Um dieses nicht deterministische Verhalten zu eliminieren empfiehlt es sich hier aus dem impliziten ein explizites ODER zu machen, indem man einen *Verzweigungsknoten* benutzt, oder eine explizite Parallelisierung mittels *Parallelisierungsknoten* zu verwenden.

Signale

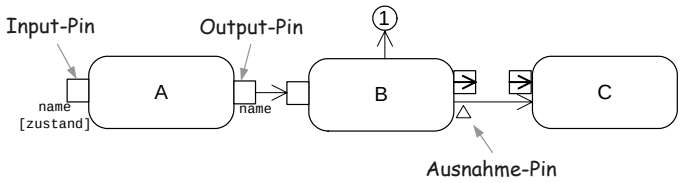
Ein weiteres Element, das in Abb. A.16 vorkommt und der Erklärung bedarf ist das *Signal*. Signale werden zur Modellierung *asynchroner* und *nebenläufiger* Nachrichten benutzt. Sie stehen damit in komplettierender Beziehung zu den *synchronen Verhaltensaufrufen*, die mit dem *Forken-Symbol* gekennzeichnet werden und somit etwa den Aufruf einer weiteren Aktivität, eines Zustandsautomaten, etwa den Aufruf einer Operation repräsentieren kann. Verhaltensaufrufe haben wir bereits im Verlaufe des Buches im Kontext der *Action-Semantics* (vgl. Kapitel 3) kennen gelernt und wollen sie daher an dieser Stelle nicht näher erläutern. *Eingehende Signale* liefern immer einen Objektknoten, der das empfangene Signal darstellt , an die adjazenten Knoten. Anders als bei Aktionen wird dies allerdings nicht durch die Darstellung von Parameter-Pins ausmodelliert.

Ausnahmebehandlung

Abschließend zu Aktivitätsdiagrammen soll nun noch kurz auf die Möglichkeit der *Ausnahmebehandlung* eingegangen werden. Tritt eine Situation auf, die das Unterbrechen der vorliegenden Tätigkeit verlangt, kann der Bereich innerhalb der Aktivität in der die Aus-

nahme auftreten kann durch eine gestrichelte Linie als *Ausnahmebereich* gekennzeichnet werden. Im Beispiel (Abb. A.16) ist dies der Fall, wenn die bestellte Ware nicht innerhalb eines bestimmten Zeitraumes geliefert werden kann. Der Zeitraum ist modelliert als *Zeitereignis*, das eine Sonderform des schon besprochenen *Signals* darstellt. Die Kante die den Ausnahmekontrollfluss modelliert, wird durch ein Blitzsymbol dargestellt. In der Black-Box Notation ist auch die Darstellung einer Ausnahmensituation innerhalb einer einzelnen Aktion darstellbar. Dies geschieht mittels eines speziellen Ausgabepins, der nur dann Daten liefert, wenn die Aktion bedingt durch eine Ausnahmesituation abgebrochen wird. Dargestellt wird dies durch das dreieckige Symbol für Ausnahmen (Abb. A.22). In der Abbildung dargestellt ist die Situation, dass der Kontrollfluss im Normalfall von Aktion `A` zu `B` verläuft und dann einen Weg nimmt, der an einer weiterer Stelle im Diagramm weiter detailliert ist und hier nicht weiter dargestellt ist. Symbolisiert wird dies durch den Kreis mit eingefasster Nummer. Der Kontrollfluss geht dann an der Stelle mit passendem Gegenstück weiter.

Abb. A.22 Ausnahmepin



Tritt jedoch während der Ausführung von `B` eine Ausnahmesituation auf, die die weitere Ausführung verhindert, wird zu Aktion `C` verzweigt, die zusätzlich als Eingabeparameter ein Datenpaket bekommt, in dem beispielsweise der Grund des Abbruchs transportiert werden könnte.

A.2.3 Zustandsautomat

Betrachten wir nun einen weiteren Vertreter der „Großen“ zur Modellierung von Verhalten: *Zustandsautomaten* beschreiben den Lebenszyklus eines `Classifiers` in Bezug auf die möglichen Zustände, die er einnehmen kann und definieren die Reihenfolge und Bedingungen in der Wechsel zwischen Zuständen geschehen können.

Zustandsautomaten werden oft dazu verwendet Objekte (als Instanzen von Klassen) näher zu beschreiben. Vor allen Dingen spielen dabei zwei Aspekte eine Hauptrolle in der Betrachtung:

- die *Attribute* des Objektes und die *Wertbelegung* dieser Attribute in einem konkreten Zustand (dazu gehören auch die Beziehungen, die ein Objekt zu einem Zeitpunkt zu anderen Elementen unterhält),
- das *Verhalten* des Objektes in Form eines *Zustandswechsels*. Hier existieren die Möglichkeit des Wartens auf ein genau spezifiziertes Ereignis – dem Trigger des Zustandswechsels – sowie das *Ausführen einer Aktion* innerhalb des erreichten Zustandes.

Abb. A.23 zeigt einen Ausschnitt der Möglichkeiten, die die UML zur Modellierung dieser Aspekte eines Systems bietet:

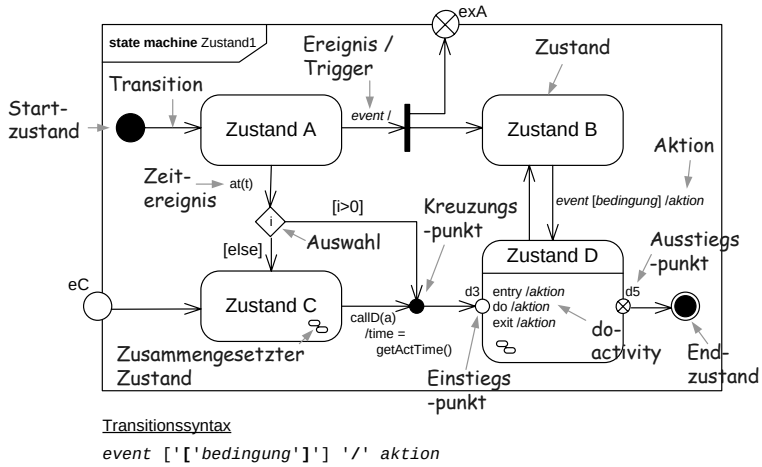


Abb. A.23 Notationen Zustands-diagramm

Ein *Zustandsautomat* besteht aus einer Menge von *Zuständen*, den *Transitionen (Übergängen)* zwischen den Zuständen, den *Triggern (Auslösern)* der Transitionen sowie der Angabe des Verhaltens, das während der Ausführung der Transition oder während der Dauer in der sich das System in einem diskreten Zustand befindet ausgeführt wird.

Zustände werden durch an den Ecken abgerundete Rechtecke dargestellt. Von ihnen ausgehende unidirektionale Pfeile zu anderen Zuständen symbolisieren die potenziellen Wechsel in andere Zustände, die in diesem Zustand möglich sind. Diese *Transitionen* können mit *Triggern* versehen werden, die das *Ereignis (engl. event)* und/oder die Bedingung enthalten können, die erfüllt sein müssen, damit die *Transition* schalten kann. Ein solches Ereignis kann beispielsweise das Empfangen eines Signals oder der Aufruf einer Operation sein. Im letzteren Fall kann das Zustandsdiagramm dazu benutzt werden das Verhalten einer Klasse in Form ihrer Operationen näher zu spezifizieren. Eine spezielle Art der Ereignisse sind die

Zeitereignisse, die das Erreichen eines speziellen Zeitpunkts (`at(`*`Zeitpunkt`*`)`) oder die Dauer eines verstrichenen Zeitintervalls (`after(`*`Dauer`*`)`) anzeigen.

Wird auf die Modellierung eines auslösenden Ereignis verzichtet („eine Transition ohne Beschriftung“), schaltet der Automat, sobald alle Aktivitäten des Zustandes – sowie alle eventuell in ihm geschachtelten Regionen – beendet sind (`Completion Event`).

Sollen schon während des Zustandsüberganges ein Verhalten (`Behavior`) ausgeführt werden, kann dessen Name, abgetrennt durch ein „/“ an die Transition angetragen werden. Auch innerhalb eines Zustandes existieren vordefinierte Punkte, an denen Verhalten anmodelliert werden kann, das dann ausgeführt wird, wenn der Zustand einen durch diesen Punkt definierten Abschnitt innerhalb seines Lebenszyklus erreicht hat. Diese vordefinierten Punkte sind:

- `entry`: Wird bei Eintritt in einen Zustand erreicht. Das hier angegebene Verhalten werden vor jeder weiteren Aufgabe ausgeführt.
- `do-activity`: Wird im Anschluss and die `entry` Phase ausgeführt. Ist das hier aufgeführten Verhalten beendet, wird ein `Completion Event` ausgelöst. Diese Verhaltensausführung kann durch andere Ereignisse unterbrochen werden, worauf die Sektion verlassen und der nächste Punkt im Lebenszyklus erreicht wird.
- `exit`: Wird unmittelbar vor dem Verlassen eines Zustandes erreicht.

Wird ein Zustand/Zustandsautomat aktiviert, startet er normalerweise im *Initialzustand*, der durch einen schwarz ausgefüllten Kreis symbolisiert wird. Prinzipiell kann jedoch jeder Zustand des Automaten bzw. eines geschachtelten Zustands zum Startzustand werden.

Zusammengesetzte Zustände

Bei diesem Sonderfall der geschachtelten Zuständen spricht man auch von *Zusammengesetzten Zuständen (engl. composite states)*. Sie werden dargestellt, indem entweder der enthaltene Zustandsautomat innerhalb des Rechtecks ausmodelliert, oder das stilisierte Symbol in Form zweier verbundener Zustände benutzt wird. Abb. A.24 zeigt ein Beispiel für einen zusammengesetzten Zustand mit zwei Regionen und ausmodellierten Unterzuständen.

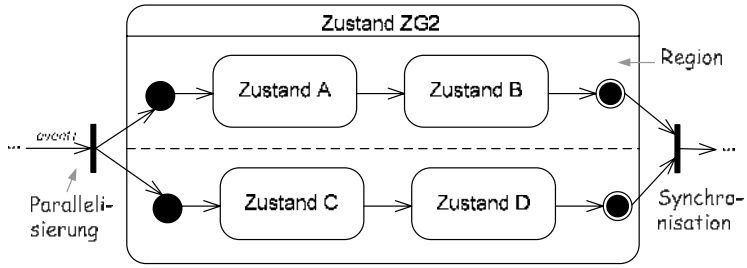


Abb. A.24 Zusammengesetzter Zustand mit Regionen

Die Transition schaltet bei Auftreten des Ereignisses *event*. Danach findet eine *Parallelisierung* statt und die Ausführung des `Zustandes ZG2` beginnt parallel in beiden *Regionen* im jeweiligen Startzustand. Generell können zusammengesetzte Zustände beliebig viele (und müssen mindestens eine) Region besitzen, deren enthaltene Zustände auch wiederum zusammengesetzt sein können, sodass die Möglichkeit einer beliebig tiefen Verschachtelung besteht. Nachdem beide Regionen abgearbeitet sind, wird im Beispiel der Kontrollfluss mittels einer *Synchronisation* wieder zusammengeführt und die Abarbeitung des nächsten Zustandes beginnt.

Ein-/Ausstiegspunkte

Abb. A.25 zeigt eine weitere Möglichkeit des Einstiegs in einen Zustand.

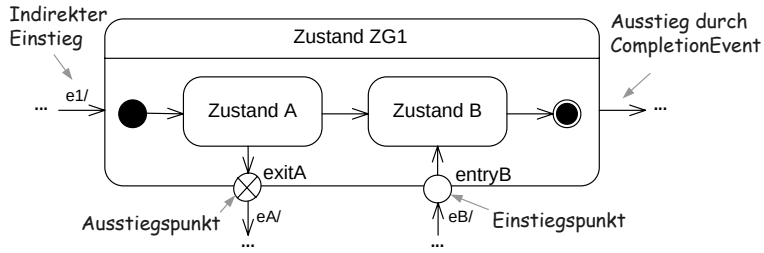


Abb. A.25 Explizite Ein-/Ausstiegspunkte

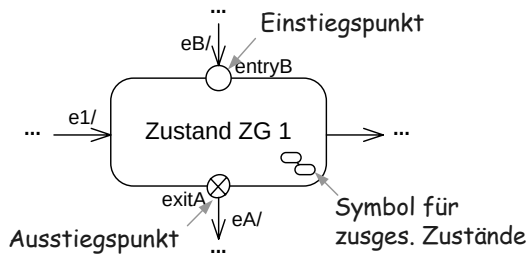
Dargestellt sind zwei Möglichkeiten des Ein-/Ausstiegs in/aus dem Zustand:

- *Indirekter Einstieg* über die Transition mit Ereignis `e1`: Da die Transition am Rand des Rechtecksymbols des Zustandes endet, wird der Standardeinstiegspunkt gewählt, der in diesem Falle dem Initialpunkt entspricht.
- *Direkter Einstieg* am *Einstiegspunkt* `entryB` über die Transition mit Ereignis `eB`.
- *Ausstieg über den Endzustand*, der erreicht wird, sobald im Zustand `B` alle Aufgaben erledigt sind.

- *Expliziter Ausstieg* über den *Ausstiegspunkt* `exitA`, der nur dann angelaufen wird, wenn Ereignis `eA` eintrifft, bevor alle Aktivitäten in `Zustand A` beendet worden sind. Ansonsten wird automatisch in `Zustand B` gewechselt.

Das Konzept der expliziten Ein-/Ausstiegspunkte wurde in Version 2 der UML neu eingeführt und unterstützt die Wiederverwendbarkeit von Zuständen in unterschiedlichen Kontexten. Alternativ zu den vorgestellten Notationen können zusammengesetzte Zustände auch in der *Black-Box Darstellung* notiert werden. Dazu wird am unteren Rand ein stilisiertes Symbol zweier verbundener Zustände verwendet. Abb. A.26 zeigt den eben besprochenen `Zustand ZG1` in dieser Darstellung.

Abb. A.26 Alternative Notation zusammengesetzer Zustände



Abschließend noch die Erläuterung der bisher noch unerwähnten Elemente aus Abb. A.23:

Auswahl und Zusammenführung

Der Kontrollflusssteuerung dienen: *Auswahlknoten (engl. choice state)*, die die Steuerung des Flusses in Abhängigkeit von zuvor definierten Bedingungen erlauben und durch ein Diamantsymbol dargestellt werden. Die Bedingungen werden in eckigen Klammern an die betroffenen Transitionen angetragen. Um eine einfach Zusammenführung einer solchen Auftrennung des Flusses zu ermöglichen, wird das Element des *Kreuzungspunktes* definiert, das durch einen schwarzen Punkt symbolisiert wird und immer dann benutzt werden kann, wenn von unterschiedlichen Zuständen aus ohne weitere Bedingungen zum selben Zustand geschaltet wird.

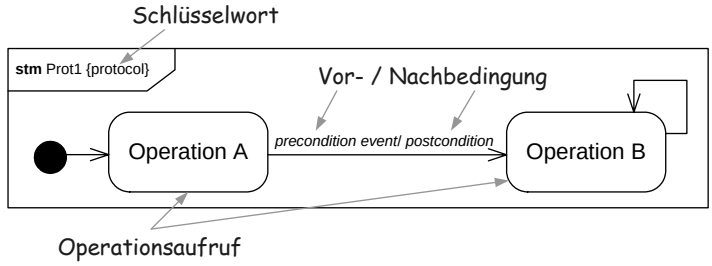
Protokollautomaten

Neben den im vorigen Abschnitt besprochenen allgemeinen Zustandsautomaten, die wie beschrieben jeglicher Art von `Classifier` zugeordnet werden können, um ihr *Verhalten* zu Beschreiben, existiert noch eine Spezialisierung dieser Automaten innerhalb der UML2, die speziell für die Modellierung der Bedingungen ge-

schaffen wurde, die für den *Aufruf der Operationen* eines `Classifiers` gelten.

Abb. A.27 zeigt, wie ein solcher *Protokollautomat* prinzipiell aufgebaut ist. Im Wesentlichen gelten dieselben Regeln, wie in allgemeinen Zustandsautomaten.

Abb. A.27 Protokollautomat



Das Schlüsselwort `{protocol}` zeigt an, dass es sich um diese spezielle Art Automat handelt. An Transitionen werden *Vor- und Nachbedingungen (engl. pre-/postconditions)* angetragen, die erfüllt sein müssen, damit die Transition schalten kann. Zustände repräsentieren nun direkt die zugehörige *Operation*, der Wechsel in einen Zustand den Aufruf derselben. Da durch den Automaten die erlaubte Reihenfolge der Operationsaufrufe festgelegt wird, spricht man von einem *Protokoll* der Aufrufe.

Auf Grund der beschriebenen Charakteristiken ist ein Protokollautomat ideal dafür beschrieben die dynamischen Aspekte von Schnittstellen wie Interfaces oder Ports zu beschreiben

A.2.4 Sequenzdiagramm

Sequenzen gehören zu der Gruppe der so genannten *Interaktionen.* Insgesamt gehören folgenden Diagrammarten zu den *Interaktionsdiagrammen*:

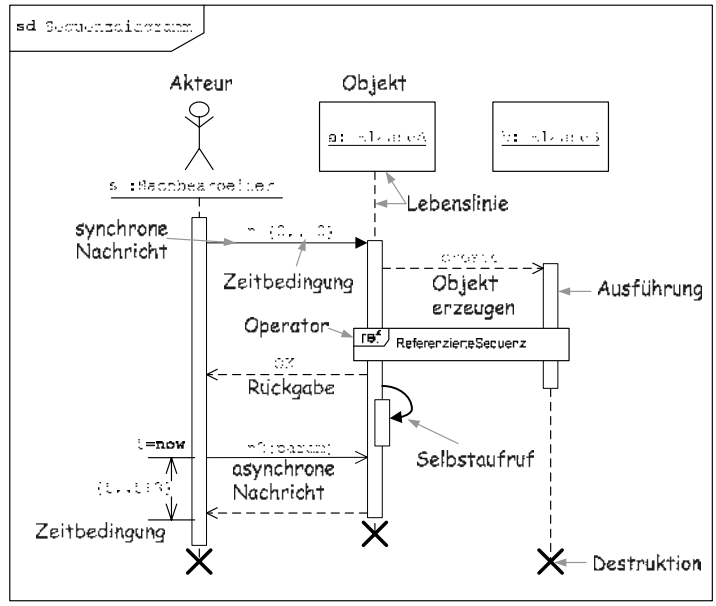
- das *Sequenzdiagramm* (behandelt in Abschnitt A.2.4),
- das *Kommunikationsdiagramm* (behandelt in Abschnitt A.2.6),
- das *Interaktions-Übersichts* Diagramm (behandelt in Abschnitt A.2.5),
- sowie das *Timing Diagramm* (behandelt in Abschnitt A.2.7)

Sequenz-diagramm

Interaktionen beschreiben eine zeitliche Abfolge von Ereignissen. Im Falle des Sequenzdiagramms bedeutet dies konkret die Beschreibung des Nachrichtenaustausches zwischen Elementen des Systems zur Laufzeit, wobei besonderer Augenmerk auf die zeitliche Abfolge des Versendens und Empfangens dieser Nachrichten gelegt wird. Im Diagramm werden die kommunizierenden Elemente dargestellt als so genannte *Lebenslinien (engl. lifelines)*.

Abb. A.28 zeigt ein Beispiel, das die meistgenutzten Elemente dieser Diagrammart enthält:

Abb. A.28 Notation Sequenz-diagramm



Nachrichtensyntax:
<attribute> '=' <signal-oder-operationsname> ['(' [<argument> [','<argument>]*] ')'] [':' <rückgabewert>]

Elemente des Diagramms (Kürzel ist das Präfix **`sd`**), die Nachrichten austauschen, müssen vom Typ `NamedElement` sein und werden mit ihrem zugehörigen `Classifier`-Symbol dargestellt an das eine gestrichelte Linie angetragen wird. Diese – nach unten verlängert – repräsentiert die *Lebensdauer* des Elementes. Die Lebensdauer umfasst den Zeitraum von der *Erzeugung* eines Objektes bis zu dessen *Destruktion*.

Im Beispiel ist der `Sachbearbeiter s1` von Anfang an existent, während die Instanz `b` von `Klasse B` erst zur Laufzeit erzeugt wird. Die Erzeugung zur Laufzeit muss von einem bereits existierenden Element angestoßen werden und geschieht mittels der `create`-Beziehung. Wird an die Lebenslinie zusätzlich von einem

Rechteck „ummantelt“ bedeutet dies, das sich das zugehörige Elemente in der *Ausführung* befindet. Die Darstellung dieses Sachverhalts ist optional.

Da ein Geschehen zur Laufzeit dargestellt wird, werden Instanzen modelliert für die zwei sinnvolle Hauptgruppen in Frage kommen: Instanzen von *Akteuren* und *Objekte* (Instanzen von Klassen). Wie bereits angeklungen bedeutet die Verlängerungen der Lebenslinien (bzw. der zugehörigen gestrichelten Linien) nach *unten* einen Zeitverlauf in die Zukunft. Je höher also ein Ereignis auf der Lebenslinie zu finden ist (Nachricht empfangen/senden), desto jünger ist es.

Nachrichtenaustausch

Das *Versenden* einer Nachricht wird durch einen ausgehenden Pfeil an der Lebenslinie des sendenden Elementes hin zur Lebenslinie des empfangenden Elementes repräsentiert. Pfeile mit ausgefüllter Pfeilspitze stellen *synchrone Aufrufe* dar (das sendende Objekt wartet mit weiteren Aktivitäten bis die aufgerufene Interaktion beendet ist) während „hohle“ Pfeilspitzen *asynchrone Aufrufe* symbolisieren.
Dabei kann es durchaus vorkommen, dass sich Pfeile überkreuzen (in der Abbildung nicht dargestellt).

Die *Beschriftung der Nachrichten* umfasst den (qualifizierten) *Namen* der Nachricht, sowie eventuell mitgegebene *Parameter* und, ebenso optional, den *Rückgabewert* des Aufrufes. Die konkrete Beschreibung der Nachricht unterscheidet sich durch den Typ der Nachricht. Grundsätzlich existieren zwei Nachrichtenarten:

- *Operationsaufrufe* und
- *Signale*.

Handelt es sich bei der modellierten Nachricht um einen Operationsaufruf, ist darauf zu achten, dass die Beschriftung kompatibel zur *Signatur* einer Operation ist, die innerhalb der die Nachricht empfangenden Seite definiert ist. Bei *Signalen* ist auf die Kompatibilität eventuell mitgegebener *Argumente* zu achten. Ist eine aufgerufene Interaktion beendet kann dies durch einen *Rückgabe* Pfeil in Richtung `Empfänger der ursprünglichen Nachricht -> Sender` modelliert werden. Falls gewünscht kann hier auch der tatsächliche Rückgabewert angetragen werden.

Referenzen und Fragmente

Durch die Nutzung des *Referenzierungs-Konstruktes* können Sequenzen beliebig tief ineinander geschachtelt werden. Dazu wird ein Diagramm-Rahmen mit „*Operator*“ `ref` auf die betroffenen Le-

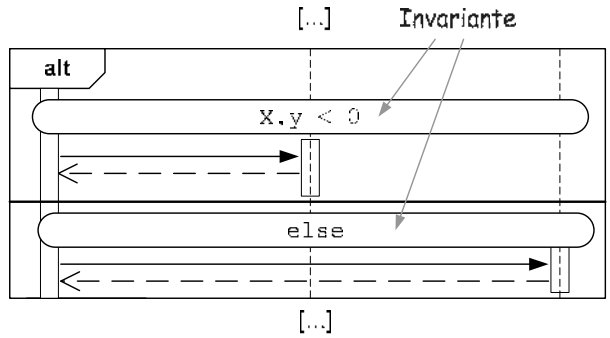
benslinien gelegt und der Name der aufgerufenen Sequenz, in die Mitte des Rahmens geschrieben (vgl. Beispiel Abb. A.28). Zur Beeinflussung des Kontrollflusses der Interaktion stehen so genannte *Kombinierte Fragmente (engl. CombinedFragments)* zur Verfügung, die durch weitere Operatoren repräsentiert werden und hier nur kurz aufgeführt werden sollen:

Abb. A.29 Zulässige Operatoren für Fragmente

Operator	Schlüsselworte	Beschreibung
alt	[Bedingung x] [Bedingung y] [else]	Ausführung einer bestimmten Sequenz abhängig von der Auswertung einer oder mehrerer Bedingungen.
opt	[Bedingung x]	Die Sequenz wird nur ausgeführt, wenn die Bedingung erfüllt ist.
par		Die enthaltenen Sequenzen werden nebenläufig (parallel) ausgeführt.
loop	min, max, [Bedingung x], break	Stellt eine Schleife dar. Die Sequenz wird mindestens `min` aber maximal `max` mal ausgeführt. Alternativ kann auch `while x` oder `until x` verwendet werden. `break` beendet den Loop zu jedem beliebigen Zeitpunkt.

Abb. A.30 zeigt einen Ausschnitt aus einem Sequenzdiagramm, das den `alt` Operator anwendet.

Abb. A.30 `alt` Operator



Die *Invariante* stellt jeweils die Bedingung dar, die erfüllt sein muss, damit das jeweilige Fragment ausgeführt werden kann. Ist der Wert der Variablen `y` von Instanz `X` kleiner Null, kommt das obere Fragment zum Zug, in allen anderen Fällen entsprechend das untere. Die umgebende Sequenz, in die das kombinierte Fragment eingebettet

ist, ist nicht dargestellt. Die Einbettung sieht jedoch genau so aus, wie die Einbettung der Referenzierung in Abb. A.28.

Zeitbedingungen

Als letztes Detail zu Sequenzdiagrammen wollen wir noch die Möglichkeiten vorstellen, die die UML2 zur Modellierung von Bedingungen an den zeitlichen Ablauf innerhalb einer Sequenz bietet. Diese *Zeitbedingungen* ermöglichen es uns beispielsweise die maximale Dauer vom Senden einer Nachricht bis zum Empfang der Antwort zu spezifizieren, aber auch generell die maximale Zeitdauer zwischen zwei beliebigen Punkten auf Lebenslinien festzulegen. So legt Abb. A.28 beispielsweise die Dauer von Nachricht `n1`, die `Sachbearbeiter s1` an Objekt `a` sendet, auf einen Zeitraum zwischen `0` und `10` *Zeiteinheiten* fest.

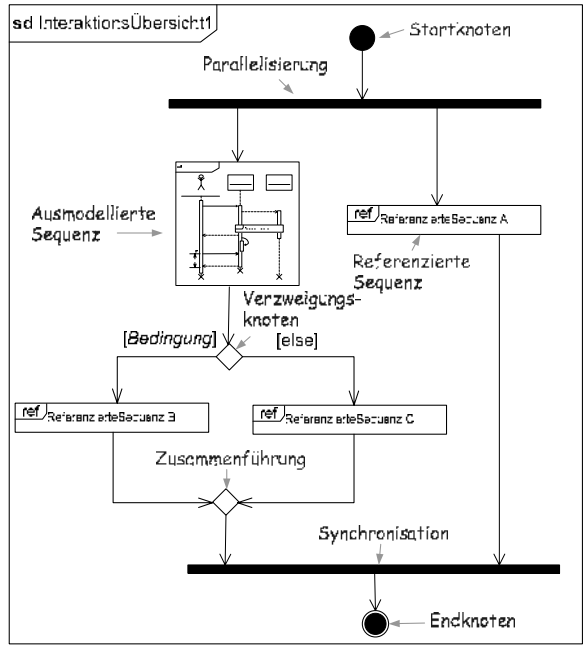
Die erlaubten Intervalldauern können auch Variablen enthalten. Im selben Beispiel wird die Variable `t` mit dem aktuellen Zeitstempel (Schlüsselwort `now`) initialisiert und dann festgelegt, dass nach dem Absenden von Nachricht `n2` von `s1` an `a` bis zum Eintreffen der Antwort höchstens der Zeitpunkt `t+3` erreicht sein darf.

A.2.5 Interaktions-Übersichts Diagramm

Während die im vorigen Abschnitt beschriebenen Sequenzdiagramme jeweils immer nur ein Szenario pro Diagramm darstellen können – einen konkreten aus allen möglichen Abläufen des Verhaltens – versucht das *Interaktions-Übersichts Diagramm* das Gesamtverhalten im Auge zu behalten, indem es mit den Mitteln der Kontrollflusssteuerung des Aktivitätsdiagramms (vgl. A.2.2) verschiedene Interaktionen zu einem Gesamtfluss kombiniert. Dazu werden die in einen Zusammenhang gestellten Interaktionen entweder ausmodelliert dargestellt, oder im Falle der Sequenzen etwa das in A.2.4 beschriebene Konzept der *Referenzierung* benutzt.

Aufgrund der Seltenheit dieser Diagrammart wollen wir uns auf die beispielhafte Darstellung der möglichen Konstrukte in Abb. A.31 beschränken und auf weitere Erklärungen auf die Erklärung der Kontrollflusselemente im Abschnitt über Aktivitätsdiagramme im entsprechenden Abschnitt (s. o.) verweisen.

Abb. A.31 Notation Interaktions-Übersichten



A.2.6 Kommunikationsdiagramm

Kommunikationsdiagramme (engl. communication diagrams) entsprechen vereinfachten Sequenzdiagrammen und konzentrieren sich auf die Darstellung auf die Interaktion (den Nachrichtenaustausch) zwischen *Lebenslinien* (vgl. Sequenzdiagramme in Abschnitt A.2.4) und die dazu notwendige architekturelle Struktur des betrachteten Systems. Wie auch bei Sequenzdiagrammen findet die Betrachtung der Elemente zur Laufzeit statt und betrachtet nicht die Gesamtheit aller möglichen Abläufe innerhalb des Systems, sondern nur ein konkretes Szenario.

Anders als bei Sequenzdiagrammen wird die Reihenfolge der versendeten Nachrichten nicht durch die Anordnung auf einer vertikalen Achse, sondern durch eine konsistente, hierarchische Benennung der Nachrichten (üblicherweise eine Durchnummerierung) mit so genannten *Sequenztermen* dargestellt. Eine Folge dieser Darstellungsweise ist, dass der Sachverhalt des „gegenseitigen Überholens“ von Nachrichten *nicht* darstellbar ist: Wird `Nachricht A` an einem früheren Zeitpunkt an Lebenslinie `C` gesendet als `Nachricht B` trifft `B` jedoch früher ein als `A`, kann dies nicht ohne weiteres modelliert werden. Zur weiteren Vereinfachung wird auch auf

die Möglichkeiten der Verschachtelung und starken Strukturierung durch kombinierte Fragmente oder ähnlicher Konstrukte wie sie bei Sequenzdiagrammen möglich sind verzichtet.

Abb. A.32 zeigt die Notationselemente des Diagrammtyps:

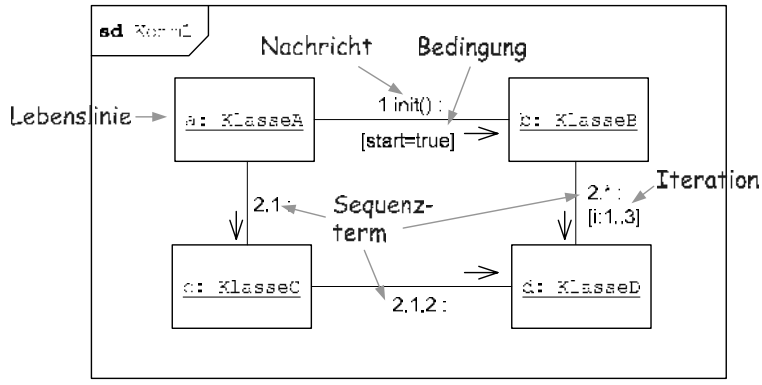


Abb. A.32 Notation Kommunikationsdiagramm

Syntax von Nachrichten:
<sequenzterm> '.' <sequenzterm> . <nachricht> ':'
<sequenzterm> ::= [*integer* *name*] [<wiederholung>]
<wiederholung> ::= '*' [*intervall*] | '||' [*intervall*] | '[' *bedingung* ']'

Der *Nachrichtenaustausch* wird durch Assoziationslinien angezeigt. Diese bezeichnen jedoch nur, dass eine Kommunikation stattfindet, die strukturelle Beziehung der beteiligten Elemente (Kompositionsbeziehung, Referenzierung, …) bleibt in der Darstellung offen und muss, falls notwendig, in einem separaten Klassendiagramm notiert werden. *Nachrichten* können mit *Namen* und *Parameterliste* (beide optional) angegeben werden und enden mit einem Doppelpunkt („:“).

Reihenfolge der Nachrichten

Wie bereits erwähnt wird die Reihenfolge des Nachrichtenaustausches durch die Verwendung von *Sequenztermen* modelliert. Diese werden üblicherweise durch eine hierarchische Durchnummerierung (wie beispielsweise die Abschnittsnummerierung dieses Buches) realisiert, wobei Nummern oder Namen auf gleicher Ebene (E1.E2.E3…) die gleiche Verschachtelungstiefe symbolisieren.

Kontrollstrukturen

Auf Nachrichtenlinien können zwei Arten von Kontrollstrukturen angewendet werden, um den Fluss der Interaktion dynamisch zu beeinflussen:

Bedingungen werden in eckigen Klammern angegeben und realisieren so bedingte Verzweigungen. Ein Nachrichtenaustausch entlang einer Bedingung kann nur dann stattfinden, wenn die Bedingung mit zu `true` ausgewertet wird.

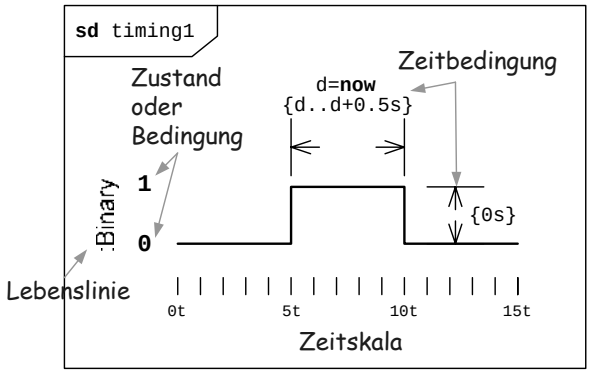
Iterationen geben eine wiederholte Ausführung einer Kommunikation wieder und entsprechen damit einem *Schleifen-Konstrukt*. Anstatt des Sequenzterms wird die Wildcard „`*`“ notiert, gefolgt vom *Wiederholungsintervall* in eckigen Klammern. Die Voranstellung des Doppelbalkens „`||`“ vor die Wildcard symbolisiert die *parallele Ausführung* der Schleife. Es werden also `n` *Threads* gleichzeitig gestartet. Wird auf diese zusätzliche Angabe verzichtet wird automatisch eine *sequenzielle Ausführung* angenommen.

A.2.7 Zeitdiagramm

Zeitdiagramme (engl. timing diagrams) werden eingesetzt, um zeitliche Aspekte in der Interaktion zwischen bzw. Zustandsänderungen innerhalb von einem oder einer Gruppe von `Classifiern` zu beschreiben. Sie finden vor allem im Bereich der Elektrotechnik und dem Bereich der eingebetteten Systeme Verwendung. Die angebotenen Notationselemente und Konzepte ähneln dabei sehr denen aus den *Sequenzdiagrammen* (vgl. Abschnitt A.2.4). Auch in diesem Diagramm werden Aspekte dargestellt, die sich zur Laufzeit des `Classifiers` abspielen. Die Instanzen der beteiligten `Classifier` heißen *Lebenslinien*.

Abb. A.33 zeigt die Modellierung eines Zustandswechsels:

Abb. A.33 Notation Zeitdiagramm – Zustandswechsel



Der Zustand der (nicht näher benannten) Lebenslinie `:Binary` beginnt zum Zeitpunkt `0t` im Zustand `0` und wechselt ohne Übergangszeit in den Zustand `1`. Dieser dauert höchstens eine halbe Sekunde und wechselt dann zurück in Zustand `0`.

Neben der Modellierung von Zustandswechseln kann im Zeitdiagramm auch die Kommunikation zwischen Lebenslinien dargestellt werden. Die Lebenslinien werden dann ähnlich dem Konstrukt der

kombinierten Fragmente des Sequenzdiagramms in eigenen Kompartments notiert und die Nachrichten als Pfeile zwischen den Zustandslinien angetragen. Auch hier existieren alle Möglichkeiten der Modellierung von Nachrichten, die auch bei Sequenzdiagrammen bestehen (synchron, asynchron, mit/ohne Parameter, …). Da diese Konzepte bereits ausführlich bei Sequenzen erörtert worden sind, werden sie hier nicht wiederholt.

A.3 Literaturtipps

Als erste Anlaufstelle, sowie als Endpunkt zur Klärung der letzten Details der UML 2 sind die originären Spezifikationsdokumente [UML2IS] und [UML2SS] der OMG zu nennen.

Wem die Darstellung dort zu trocken ist, der kann (mittlerweile) auch auf dem Buchmarkt fündig werden. Das Metamodell der UML 2 wird in der dort angebotenen Literatur allerdings bislang eher stiefmütterlich behandelt. Selbst das Referenzwerk der *Drei Amigos* [RBJ04] bildet da keine Ausnahme, ebenso das bereits erwähnte *UML Glasklar* [JRH+03] und der Klassiker [Oes04] von Bernd Oestereich. Diese Bücher bedienen die Anwenderperspektive und gehen weniger auf die *Gory Details* der UML und ihres Metamodells ein. Wer der UML wirklich auf den Grund gehen möchte, muss allerdings genau diesen Weg gehen und wird in der folgenden Auswahl fündig:

[BHK04] nimmt den Weg über die Metamodell-Sicht und schneidet zudem die Themen MOF 2 und MDA an. Ein wirklich nützliches Buch zum Thema Use Cases und Patterns, die sich im Alltag der Anforderungsaufnahme mit dieser speziellen Technik als besonders nützlich erwiesen haben, ist [GP04].

Speziell zu dem stark überarbeiteten Bereich Aktivitäten und Aktionen, gibt es eine Artikelserie von Conrad Bock ([Boc03a] - [Boc04b]), die zudem eine Einführung in das UML-Repository und die UML-ActionSemantics beinhaltet. Auf die Aspekte der Verhaltensmodellierung mittels Aktivitätsdiagrammen geht auch [PR06] ein. Lediglich auf englisch, aber hervorragend als fundiertes Werk über die UML 1.x – 2 geeignet – und unser Tipp, wenn Sie sich nur ein Buch über die UML kaufen wollen – ist die *UML Bible* von Tom Pender [Pen03], die durch Vollständigkeit und penible Detailkorrektheit auffällt und auch die Hintergründe der jeweiligen Konstrukte beleuchtet, dabei aber nie trocken oder zu theoretisch wird.

Literatur

[BHK04] Marc Born, Eckhardt Holz und Olaf Kath: *Softwareentwicklung mit UML 2 – Die neuen Entwurfstechniken UML 2, MOF 2 und MDA*. Addison-Wesley, 2004.

[Boc03a] Conrad Bock: *UML 2 Activity and Action Models*. In: Journal of Object Technology, Volume 2, Number 4, S. 43-53, ETH Zürich, Juli/August 2003.
http://www.jot.fm/issues/issue_2003_07/column3/ (letzter Abruf Mai 2005)

[Boc03b] Conrad Bock: *UML 2 Activity and Action Models Part 2 – Actions*. In: Journal of Object Technology, Volume 2, Number 5, S. 41-56, ETH Zürich, September/Oktober 2003.
http://www.jot.fm/issues/issue_2003_09/column4/ (letzter Abruf Mai 2005)

[Boc03c] Conrad Bock: *UML 2 Activity and Action Models Part 3 – Control Nodes*. In: Journal of Object Technology, Volume 2, Number 6, S. 7-23, ETH Zürich, November/Dezember 2003.
http://www.jot.fm/issues/issue_2003_11/column1/ (letzter Abruf Mai 2005)

[Boc04a] Conrad Bock: *UML 2 Activity and Action Models Part 4 – Object Nodes*. In: Journal of Object Technology, Volume 3, Number 1, S. 27-41, ETH Zürich, Januar/Februar 2004.
http://www.jot.fm/issues/issue_2004_01/column3/ (letzter Abruf Mai 2005)

[Boc04b] Conrad Bock: *UML 2 Activity and Action Models Part 5 – Partitions.* In: Journal of Object Technology, Volume 3, Number 7, S. 37-56, ETH Zürich, Juli/August 2004.
http://www.jot.fm/issues/issue_2004_07/column4/ (letzter Abruf Mai 2005)

[GP04] Gunnar Overgaard und Karin Palmkvist: *Use Cases : Patterns and Blueprints.* Addison-Wesley, 2004.

[JRH+03] Mario Jeckle, Chris Rupp, Jürgen Hahn, Barbara Zengler und Stefan Queins: *UML 2 glasklar*. Hanser, 2003.

[Oes04] Bernd Oestereich: *Analyse und Design mit UML 2 – Objektorientierte Softwareentwicklung, Siebte Auflage*. Oldenbourg, 2004.

[Pen03] Tom Pender: *UML Bible*. John Wiley & Sons, 2003.

[PR06] Daniel Pieper und Carsten Röttgers: *MDA mit der UML 2.0 – Neue Möglichkeiten der Verhaltensmodellierung.* In: OBJEKTspektrum 01/2006, S. 46-51. SIGS-DATACOM, Januar 2006.

[RJB04] James Rumbaugh, Ivar Jacobson und Grady Booch: *The Unified Modeling Language Reference Manual, Second Edition*. Addison-Wesley, 2004.

[UML2IS] Object Management Group (OMG): *UML 2.0 Infrastructure Final Adopted Specification*.
http://www.omg.org/docs/ptc/04-10-14.pdf (letzter Abruf November 2005)

[UML2SS] Object Management Group (OMG): *UML 2.0 Superstructure Final Adopted Specification.*
http://www.omg.org/docs/ptc/05-07-04.pdf (letzter Abruf November 2005).

B OOGPM

B.1 Einleitung und Übersicht

Die Modellierung ist seit jeher unverzichtbares Hilfsmittel bei der Analyse und der Verbesserung von Geschäftsprozessen. Leider fanden die so entstandenen Modelle und Ergebnisse bislang nur selten bis gar keine Verwendung innerhalb der modell-getriebenen Entwicklung von Anwendungen, die diese Prozesse unterstützen oder (teilweise) automatisieren sollen.

Dieser Umstand ist vor allem der Inkompatibilität der Werkzeuge sowie der damit verbundenen Unterschiedlichkeit der verwendeten Notationen geschuldet, die im besten Falle von einigen wenigen Vertretern beider Seiten gleichermaßen beherrscht werden. So entsteht eine Kommunikationslücke, die ein Konvergieren von Geschäfts- und Entwicklungswelt in weiter Ferne erscheinen lässt.

Mit der UML steht nun eine Sprache zur Verfügung, die zur *Lingua Franca* beider Seiten werden könnte. Sie stützt sich auf einige wenige Sprachkonstrukte ab, die leicht zu erlernen sind und auf die eigenen Bedürfnisse angepasst werden können. Zusammen mit der seit Version 2.0 vorhandenen Präzision der Notation erfüllt sie somit alle Anforderungen an eine Domänen-Spezifische Sprache, die die eben bedauerte Lücke überbrücken und die Ergebnismodelle der Geschäftsprozessanalyse zu vollwertigen Artefakten in der Softwareentwicklung machen könnte.

Mit der *Objektorientierten Geschäftsprozessmodellierung (OOGPM)* [OWS+03] liefern Oesterreich, Weiss, Schröder, Weilkiens und Lenhard nun nicht nur eine solche Erweiterung der UML um Notationen zur Modellierung von Geschäftsprozessen, sondern geben gleichzeitig eine Methodik zur Anwendung derselben sowie Vorgehensanweisungen, um zu den gewünschten Ergebnissen zu gelangen.

Der vorliegende Anhang soll einen ersten Eindruck der OOGPM und vor allem deren Bestandteil der Modellierung von Ergebnissen

mit Mitteln der UML vermitteln. Es ist nicht das Ziel – und kann es aufgrund des beschränkten Platzes auch nicht sein – die Methodik vollständig wiederzugeben. Wer sich näher mit der angerissenen Thematik beschäftigen will, sei auf die angebene Literatur am Ende des Anhangs verwiesen. Auch im Fallbeispiel des Buches ab Kapitel 7 werden Teile der Methodik verwendet.

Zuerst wird ein *Birds-View* über das Gesamtvorhaben gegeben, danach werden einige ausgewählte Aktivitäten und ihre Ergebnistypen anhand eines kurzen Beispiels vorgestellt und erläutert. Die Auswahl erfolgte vor allem nach dem bereits erwähnten Gesichtspunkt die Darstellung der Ergebnisse mittels UML hervorzuheben und weniger nach verfahrenstechnischer Korrektheit oder chronologischem Vorgehen.

Abb. B.1 zeigt den Gesamtablauf der OOGPM mittels eines Modells, das ebenfalls aus dem Fundus der Ergebnistypen der Methodik selbst stammt. Dargestellt wird der Geschäftsprozess *Geschäftsprozessmodellierung*. Als Modell wird ein Aktivitätsdiagramm verwendet (in der Methodik Ablaufdiagramm genannt). In ihm ist die Reihenfolge der im Prozess enthaltenen Einzelaktivitäten dargestellt, die wiederum durch die Aktionen der Aktivität repräsentiert werden.

Legende

Im Diagramm sind Abkürzungen enthalten, die wir zum Verständnis der Abbildung kurz auflösen möchten:

- **GAF** Geschäftsanwendungsfälle
- **GP** Geschäftspartner
- **GM** Geschäftsmitarbeiter
- **SAF** Systemanwendungsfälle
- **Orga.** Organisatorisch

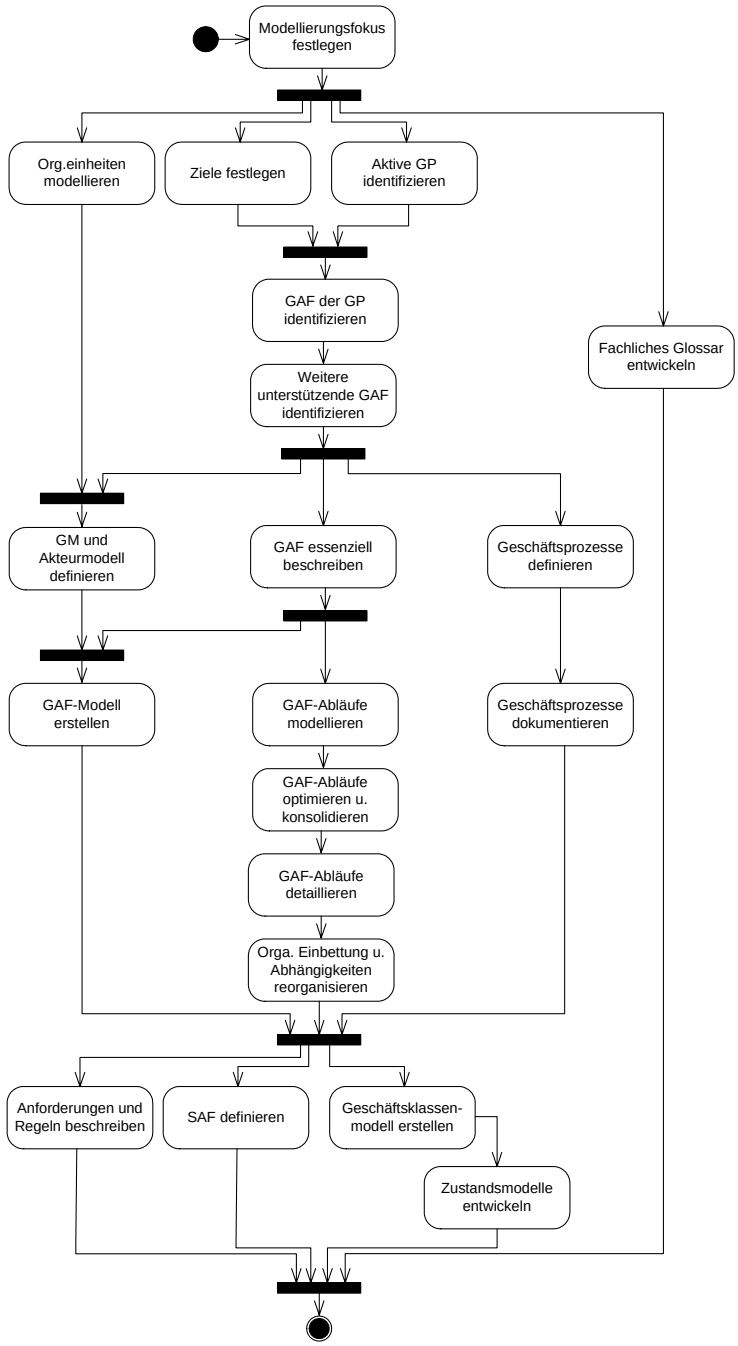
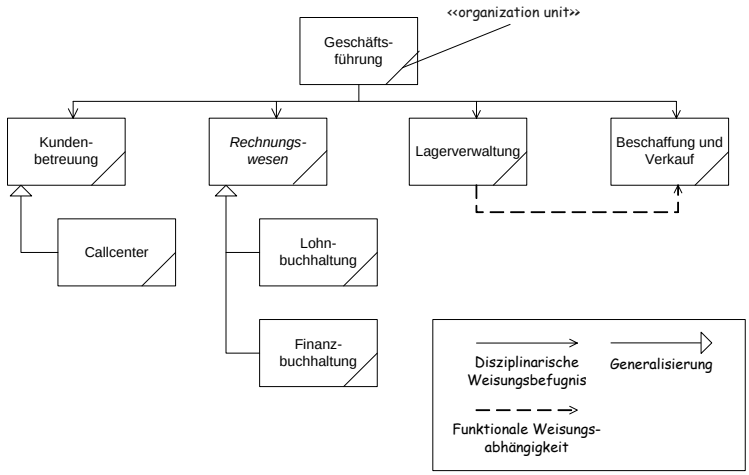


Abb. B.1 Übersicht über die OOGPM-Methodik

B.2 Organisationseinheiten modellieren

Abb. B.2 zeigt die Darstellung eines Organigramms in OOGPM-Notation. Wie man sieht, benutzt das OOGPM-Profil ein Klassendiagramm als Ausgangspunkt und erweitert lediglich die Bedeutung der verwendeten Konstrukte.

Abb. B.2 Beispiel: Organigramm



Organisationseinheiten (OE) selbst werden durch Klassen repräsentiert, die mit dem Stereotypen «`organization unit`» markiert werden. Die alternative (und in der Abbildung auch verwendete) Darstellung erfolgt durch die Verwendung eines Querstriches in der rechten unteren Ecke der Klasse, ein Symbol, das uns im weiteren Verlauf noch öfter begegnen wird.

Abstrakte Organisationseinheiten

Werden OEs als abstrakt definiert (kursiver Klassenname bzw. Eigenschaftswert {`abstrakt`}), ist dies so zu interpretieren, dass im Sprachgebrauch der modellierten Organisation zwar der Oberbegriff, jedoch keine entsprechende real existierende physische Einheit existiert. Abstrakte Organisationseinheiten müssen mittels *Generalisierung* (Vererbung, gekennzeichnet durch geschlossene Pfeilspitze) näher spezifiziert werden. Der Vererbungspfeil ist in Pfeilrichtung als „*A ist eine Art B*“ zu lesen und taucht in unserem Beispiel auf als „*Die Lohnbuchhaltung ist eine Art Rechnungswesen*“. Wie die obige Abbildung zeigt, ist die Anwendung dieses Konstruktes jedoch nicht auf abstrakte Organisationseinheiten beschränkt. So hat ein `Call-Center` alle Aufgaben der `Kundenbetreuung`, erweitert diese jedoch um die Eigenheiten, die die telefonische Annahme von Kundenanfragen mit sich bringt.

Weisungsbefugnis

Gerichtete Assoziationen stellen *Disziplinarische Weisungsbefugnis* („*A weist B an*“; „*A ist für B verantwortlich*“) zwischen Organisationseinheiten dar. Abhängigkeitsbeziehungen (gestrichelter Pfeil mit offener Spitze) stellen die gegenteilige Beziehung, also die *Funktionale Weisungsabhängigkeit* („*B ist funktional weisungsabhängig von A*“) dar. Eine Organisationseinheit, die einer anderen funktional weisungsbefugt ist, ist zwar fachlich weisungsbefugt, jedoch nicht disziplinarisch vorgesetzt.

B.3 Aktive Geschäftspartner identifizieren

Im Schritt *Aktive Geschäftspartner (AGP) identifizieren* werden nur die Teilnehmer am Prozess betrachtet, die

- *aktiv* Geschäftsprozesse initiieren und
- *außerhalb* des Modellierungsbereiches (des betrachteten Unternehmens) stehen.

Mitarbeiter unseres Beispielunternehmens fallen also durchs Raster – sie befinden sich innerhalb des modellierten Kontextes. Ebenso *nicht* betrachtet werden beispielsweise die Mitarbeiter des in Anspruch genommenen Paketdienstes. Diese werden zwar im Regelfalle von Mitarbeitern des modellierten Unternehmens angerufen und holen die Pakete ab, initiieren jedoch in der Regel nicht von sich aus einen Vorgang und verletzen somit das zweite der genannten Kriterien um in die Betrachtung mitaufgenommen zu werden.

Abb. B.3 zeigt, wie das Ergebnis der Aktivität modelliert werden kann.

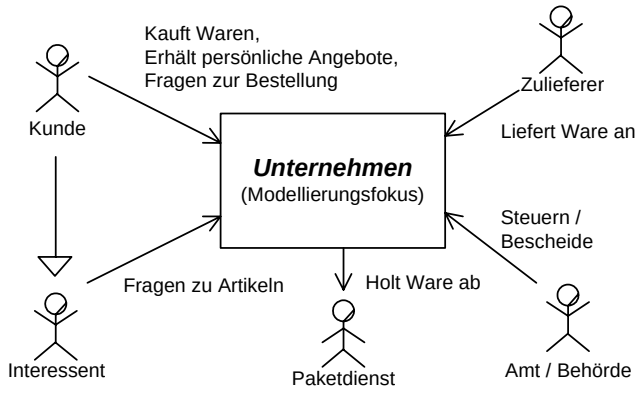


Abb. B.3 Beispiel: Aktive Geschäftspartner

AGPs werden im mittels des UML-Akteur-Symbols dargestellt. Die Zugehörigkeit zum Profil ist auch hier durch den Querstrich (am Kopf des Strichmännchens) gekennzeichnet, der hier, zusammen mit den nach oben zeigenden Armen des Akteurs, eine graphische Repräsentation des Stereotyps «AktiverGP» darstellt Die Initiativrichtung (wer stößt den GAF initial an) wird nochmals zusätzlich durch die Richtung der Assoziation zwischen Akteur und GAF dargestellt.

Sollten schon an dieser Stelle der Analyse die Rollennamen passiver Geschäftspartner auftauchen, können auch sie schon in das Modell aufgenommen werden. In unserem Beispiel ist dies der `Paketdienst`, der prompt hängende Arme und einen vom Fokus initiierten Pfeil bekommt, um den Sachverhalt der „Passivität“ zu modellieren.

B.4 Geschäftsanwendungsfälle der aktiven Geschäftspartner identifizieren

Als nächster Schritt erfolgt die Modellierung der *Geschäftsanwendungsfälle* (GAF) der aktiven Geschäftspartner (Abb. B.4). Herausgepickt betrachten wir exemplarisch die GAF der Rolle `Kunde`.

Anwendungsfallarten

Die OOGPM unterscheidet zwischen *Kern- und Unterstützenden Anwendungsfällen*. Erstere (durch einen Doppelstrich gekennzeichnet) bezeichnen Anwendungsfälle, die direkt mit dem Unternehmenszweck zusammenhängen, Letztere alle weiteren Anwendungsfälle.

Hauptziel unseres Beispielunternehmens ist der Warenverkauf. Dazu gehören direkt die Ziele der Maximierung der Kundenzufriedenheit sowie der effizienten Werbung für die verkauften Produkte. Die `Beschwerdeannahme` ist somit ein primärer Anwendungsfall, während `Kundendaten ändern` zwar eine wichtige unterstützende Funktion zur Aufrechterhaltung des Betriebes, jedoch kein primäres Geschäftsziel und damit auch keinen Kernanwendungsfall darstellt.

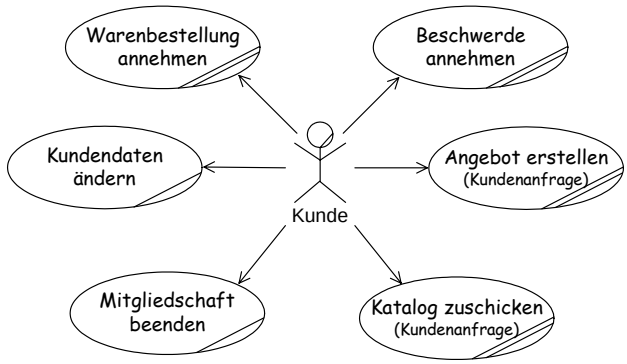


Abb. B.4 Beispiel: Geschäftsanwendungsfälle

Die Darstellung als Use-Case Modell alleine reicht natürlich zur Erfassung der GAF alleine nicht aus. Eine ausführlichere Beschreibung der Anwendungsfälle geschieht durch das Ausfüllen einer *GAF-Karte*, die an das entsprechende Symbol des Modelles innerhalb des verwendeten Modellierungswerkzeugs „angehängt" wird. An dieser Stelle der Methodik geschieht dies noch nicht besonders ausführlich (Abb. B.5), sondern liefert nur eine essenzielle Beschreibung.

Abb. B.5 Beispiel: GAF Karte

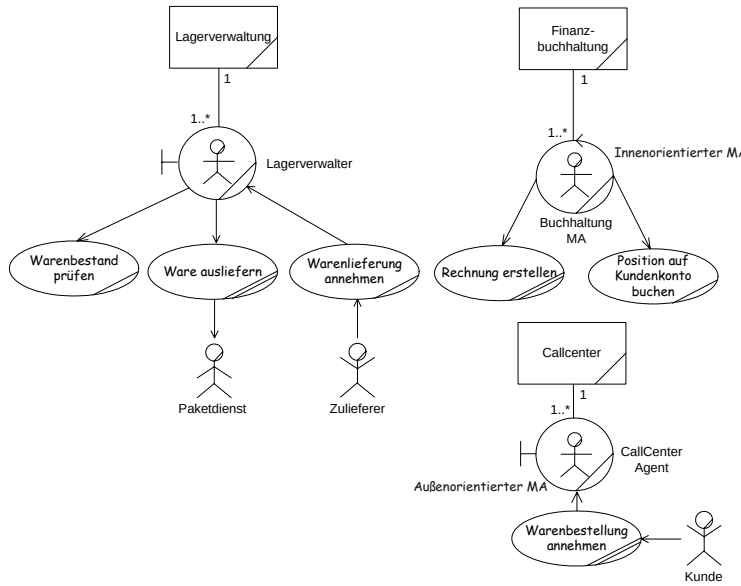
Beschreibung Geschäftsanwendungsfall	
Name	**Warenbestellung annehmen**
Art	Kern [x] Unterstützend [ ]
Kurzbeschreibung	Ein Kunde bestellt Waren.
Auslöser / Motivation	Ein Kunde wendet sich telefonisch, oder per Post an das Unternehmen.
Ergebnis	Die Waren wurden bestellt.
Akteure	Kunde

Interessant ist noch zu erwähnen, dass alle GAFs aus der Sicht des Unternehmens formuliert sind. So heißt es nicht *Angebot anfordern* (aus Kundensicht), sondern *Angebot erstellen* (Unternehmenssicht). Dies geschieht aus Gründen der Vereinheitlichung und stellt in der GAF-Modellierung ein gängiges, und auch an dieser Stelle wärmstens empfohlenes, Vorgehen dar.

B.5 Geschäftsmitarbeiter identifizieren und Akteurmodell entwickeln

Bis jetzt wurden nur die Geschäftsanwendungsfälle betrachtet, die direkt von den aktiven Geschäftspartnern initiiert wurden. Zur Abarbeitung eines kompletten Geschäftsprozesses sind jedoch weitere unterstützende GAF erforderlich, an denen nun auch passive Geschäftspartner und natürlich auch die Mitarbeiter des Unternehmens selbst beteiligt sind. Abb. B.6 veranschaulicht dies.

Abb. B.6 Beispiel: Darstellung der Querbeziehungen



Ausgehend vom Kernanwendungsfall `Warenbestellung annehmen` (unten rechts), der bereits in den vorigen Schritten ermittelt wurde, werden Beziehungen zu anderen bereits gefundenen GAF dargestellt, bzw. GAF, die erst jetzt als fehlend auffallen, in den Pool von Anwendungsfällen aufgenommen.

Mitarbeiter des fokussierten Unternehmens

Bei den Angestellten eines Unternehmens wird unterschieden zwischen *innen- und außenorientierten Mitarbeitern*, die jeweils ihr eigenes Symbol bekommen. Außenorientierte Mitarbeiter sind solche, die direkten Kontakt mit den Geschäftspartnern des Unternehmens haben, also vor allem Kunden und Lieferanten, und werden innerhalb des *Boundary Symbols* der UML dargestellt. Innenorientierte Mitarbeiter bekommen das *Control Symbol* verpasst.

Die Zugehörigkeit der Mitarbeiterrollen zu konkreten Organisationseinheiten geschieht durch Benutzung einer Assoziation, die die MAs mit den entsprechenden OEs in Beziehung setzt. Im Beispiel des Lagerverwalters ist hier angezeigt, dass genau eine `Lagerverwaltung` mit mindestens einem `Lagerverwalter` existiert.

B.6 Geschäftsprozesse definieren

Nachdem bisher vor allem statisch-strukturelle Beziehungen zwischen den modellierten Elementen hergestellt worden sind, wollen wir nun sehen, wie wir die gefundenen Anwendungsfälle einem Geschäftsprozess (GP) zuordnen sowie deren dynamisches Verhalten darstellen können.

In der folgenden Abbildung ist beispielhaft der GP Lagerverwaltung dargestellt (Abb. B.7).

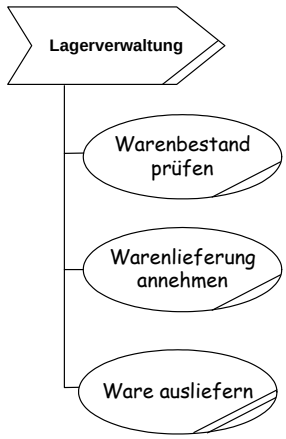


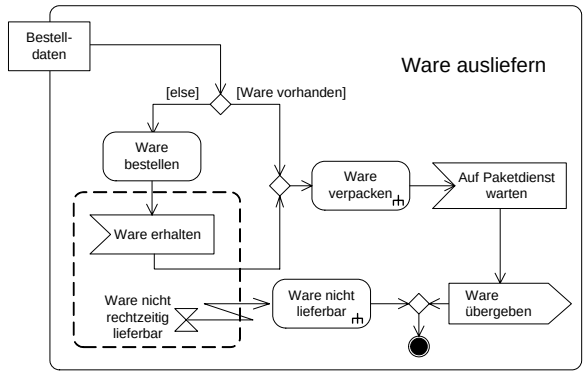
Abb. B.7 Beispiel: GP Lagerverwaltung

Geschäftsprozesse werden durch das Pfeilsymbol mit der charakteristischen Querstrichung dargestellt. Die zugehörigen Geschäftsanwendungsfälle sind durch eine ungerichtete Assoziation mit dem GP verbunden.

B.6.1 GAF-Abläufe modellieren

Abb. B.8 zeigt den dynamischen Ablauf des Anwendungsfalles `Ware ausliefern` als Aktivitätsdiagramm.

Abb. B.8 Beispiel: Aktivität Ware ausliefern



Wird der Anwendungsfall angestoßen, überprüft der `Lagerverwalter` anhand der `Bestelldaten` die Verfügbarkeit der Ware. Im Normalfall ist diese in ausreichender Zahl vorhanden, sodass diese nach Eintreffen des Paketdienstes an diesen übergeben, sowie die Übergabe zur Nachverfolgung an die zuständige Abteilung gemeldet wird.

Alternativ besteht die Möglichkeit, dass die Ware nicht in ausreichender Stückzahl im Lager vorgehalten wird. Der Lagerverwalter muss dann die Ware nachbestellen. Nicht unwahrscheinlich ist an dieser Stelle, dass die Bestellung von der Organisationseinheit Beschaffung und Verkauf erledigt wird, der dann ein entsprechender Anwendungsfall zugeordnet ist.

Trifft die nachbestellte Ware ein, kann wie im Normalfall weiterverfahren werden. Ist nach einem definierten Zeitraum die zusätzliche Ware nicht eingetroffen, wird der Vorgang der Warenauslieferung mit dem Vermerk *Nicht lieferbar* abgebrochen und eine Nachricht an die zuständige Instanz gesendet. Normalerweise wird in diesem Falle also die Abteilung, die für den Verkauf von Waren zuständig ist, die Bestellung stornieren.

Mit diesem Beispiel wollen wir unseren kurzen Ausflug in die OOGPM abschließen. Wer Appetit bekommen hat und mehr über die Möglichkeiten der Methodik und ihrer Notation erfahren möchte, den verweisen wir auf das Fallbeispiel dieses Buches oder auf die im Folgenden aufgeführte Literatur.

B.7 Literaturempfehlungen

Neben dem bereits erwähnten Buch, das die Methodik in ihrer Gesamtheit anhand eines durchgehenden Fallbeispiels beschreibt [OWS+03], bietet auch die Homepage der OOGPM einen Start-

punkt zur Vertiefung des Themas [OOGPM]. Hier finden sich neben einer Notationsübersicht auch eine erstaunlich ausführliche Leseprobe des Buches sowie viele weitere Informationen zum Thema. Im Artikel *Objektorientierte Geschäftsprozessmodellierung und modellgetriebene Softwareentwicklung* [Oes05] liefert einer der Schöpfer der Methodik einen weiteren High-Level Gesamtüberblick. Der Artikel ist auch online erhältlich und ist daher zum weiteren „Hereinschnuppern“ besonders geeignet.

Literatur

[Oes05] Bernd Oestereich: *Objektorientierte Geschäftsprozessmodelllierung und modellgetriebene Softwareentwicklung*. In: HMD – Praxis der Wirtschaftsinformatik, Heft 241, S. 27-34, dpunkt, Februar 2005.
http://www.oose.de/downloads/HMD_241_GPM_und_MDSE_Oestereich.pdf (letzter Abruf Mai 2005)

[OOGPM] oose.de: *Objektorientierte Geschäftsprozessmodellierung mit der UML – Die Homepage zum Buch*.
http://www.oogpm.de (letzter Abruf November 2005)

[OWS+03] Bernd Oesterreich, Christian Weiss, Claudia Schröder, Tim Weilkiens und Alexander Lenhard: *Objektorientierte Geschäftsprozessmodellierung mit der UML*. dpunkt, 2003.

Index

«auto-generated» 384
«entity» 380
«Pageflow» 334
«system support» 126
«ViewCall» 334
«SystemCall» 334
«StripModel» 335
«WorkflowStep» 357
«Workflow» 357
«SystemCall» 377
«ID» 380
«value» 380
«hibernate-entity» 383
«hibernate-component» 383
«HibernateSession-Factory» 386

A

Abbildungsregel *Siehe* Transformationsregel
Abstract Syntax Graph 154
Abstract Syntax Tree 169
Abstrakter Syntax Baum *Siehe* Abstract Syntax Tree
Abstraktion 61, 151
Agile MDA 234
Agiles Manifest 211
Aktion 105
  ActionInputPin 342
  AddStructuralFeature-ValueAction 104
  Pull-Semantik 342
Alexander, Christopher 37
AMM *Siehe* Architecture Metamodel
Ant 408
AOP *Siehe* Aspektorientierte Programmierung
Apache Beehive 305
Application Engineering 44, 185
Architecture Metamodel 121, 130
Architektenturm 240
Architektur 23, 130
  Beschreibung 24
  Metamodell *Siehe* Architecture Metamodel
  -muster 38
  -zentrierung 39
Artefakt 185
ASG *Siehe* Abstract Syntax Graph
Aspekt 24, 59, 126, 128
  -orientierte Programmierung 40
  -orientierung 40
AST *Siehe* Abstract Syntax Tree
ATL *Siehe* ATLAS-Transformation Language
ATLAS 390
ATLAS-Transformation Language 167, 390
Ausgang *Siehe* Outcome

B

Backus-Naur-Form
  Erweiterte 169
Beehive 305

Controls 371
System Controls 377
Beehive Page Flows
in genua 329
Belang *Siehe* Aspekt
Best Practice 185
Bock, Conrad 104
Boilerplate *Siehe* Template
Booch, Grady *Siehe* Die drei Amigos
Business Analyst 207
Business Model 123, 250
Business Unit 220
Business-Case 192, 421

C

CASE 50
CIM *Siehe* Computation Independent Model
`Classifier` 83
Client-/Benutzer-Interaktion 311
CMOF *Siehe* Complete MOF
Cockburn, Alistair 227, 234
Common Public License 280
Communication Gap 122
Compiler 32
Modell- 33
Complete MOF 88
Computation Independent Model 27, 122
Concern *Siehe* Aspekt
Constraints 97
Continuous Integration 228
Convergent
-Architecture 42
-Engineering 41
Crystal Method 227

D

Darstellungsumformung 152
Datenbindung 338
Deserialisierung 403
Diagramm 99
Dialognavigation 311
Die drei Amigos 75
Domain Driven Design (DDD) 381
Domain Engineering 43, 186
Domain Engineering Unit 222
Domain-Driven Design 135
Domain-Specific Language 70
Domäne 25
Analyse 43, 196
Entwurf 44
horizontale 129
Implementierung 44
Sub- 129
Tipps zur Analyse 196
vertikale 129
Domänen Experte 202
Domänenmodell 43, 122
DSL *Siehe* Domain-Specific Language

E

EBNF *Siehe* Backus-Naur-Form, Erweiterte
Eclipse 279
Series (Bücherreihe) 283
Tools Projekt 282
Eclipse Modeling Framework 291
Eclipse-Architektur 280
Economic Model 193
Ecore 292
EMF *Siehe* Eclipse Modeling Framework
EMOF *Siehe* Essential MOF
Entwicklungsprozess 185
Software- 185
Essential MOF 88
Evans, Eric 135, 381
Executable UML 35
Exploratory 360° 269

F

FAST 185
First-Class Extension Mechanisms 94
Forward Engineering 174
Fowler, Martin 263
Full Round-Trip Engineering 177

G

GAF *Siehe* Geschäftsanwendungsfall
Gallup Studie 422
gATL *Siehe* genua
gBPfP *Siehe* genua
GEF *Siehe* Graphical Editing Framework
Gemeinsamkeiten-Variabilitäten Analyse 190
Generative Programming 45
genua 273
  ATLAS-Transformation Language (gATL) 390
  Basisarchitektur 327
  Beehive Pageflow Profile (gBPfP) 345
  `genuacs` 332
  Hibernate Persistence Profile (gHPP) 382
  jBPM Profil 360
  JSF Metamodel (gJSFMM) 353
  Model2Model-Transformator (gM2M) 389
  Model2Text-Transformator (gM2T) 411
  Platform Independent Pageflow Profile (gPIPfP) 331
  Platform Independent Persistence Profile (gPIP) 379
  Platform Independent Workflow Profile (gPIWfP) 356
Geschäftsanwendungsfall 254
  Ablauf 261
  essenzielle Beschreibung 259
  Kern- 254
  Unterstützender 254
Geschäftsklassenmodell 263
Geschäftsmitarbeiter 256
Geschäftspartner 253
  aktiver 253
  passiver 253
Geschäftsprozess
  Ablauf 262
Geschäftsprozesse 258
Gesichtspunkt 24
gHPP *genua*
gJSFMM *Siehe* genua
gM2M Siehe genua
gM2T Siehe genua
gPIP *Siehe* genua
gPIPfP *Siehe* genua
gPIWfP Siehe genua
Grammatik
  kontextfreie 169
Graphical Editing Framework 289
Guillemets 96

H

Hay, David 263
Hibernate 283
  in genua 329
Hollywood-Prinzip 175
Human-Usable Textual Notation 32
HUTN *Siehe* Human-Usable Textual Notation

I

IBM 279
Idiom 38
IEEE1471 23
IID *Siehe* Softwareentwicklung
IMR *Siehe* In Memory Representation
In Memory Representation 154, 169
InfrastructureLibrary 81
Iteration 223
  Aufbau von 224
  -splan 278
  -sziele 225

J

Jacobson, Ivar *Siehe* Die drei Amigos
Java Emitter Templates 413
JavaCC 395
JavaServer Faces 300
  in genua 329
JBoss jBPM 312
  in genua 329

jBPM *Siehe* JBoss jBPM
JET *Siehe* Java Emitter Templates
JPF (Java Pageflow) *Siehe* Beehive
JSF *Siehe* JavaServer Faces
Jython 392
  Code-Snippets 409

K

Kleppe, Anneke 114
Konvergenz 41

L

Larman, Craig 263
Lebenszyklus von MDA-Modellen 120
Legostein-Metapher 36
Lexikalische Analyse 394
Line-printer 171

M

M2M *Siehe* Transformation, Modell-zu-Modell
M2T *Siehe* Transformation, Modell-zu-Text
Maintainer 210
Mapping
  Objekt-Relationales 283
Mapping Rule *Siehe* Transformationsregel
MDA *Siehe* Model-Driven Architecture
MDA-light 178
Meta Object Facility 31, 76, 81, 87
Metamodell 25, 84
  leichtgewichtige Erweiterung von 25
Metasprache 84
Methoden Kriege 73
Methoden Spezialist 205
Methodenkriege
  Wiedergeburt der 437
Microsoft 46
Migration 151
Milestones 229
Model-Driven Architecture 21
  agile 234
  Einführung ins Unternehmen 211
  -light 49
  Ziele der 21
Modell 24
  ausführbares 35
  berechnungs-unabhängiges (CIM) 27
  deskriptives 20
  formales 65
  -getriebene Ansätze 19
  plattform-spezifisches (PSM) 27
  plattform-unabhängiges (PIM) 27
  präskriptives 20
  -Transformation *Siehe* Transformation
Modell des Geschäftssystems *Siehe* Business Model
Modell-Fassaden 414
Modellierungsfokus 251
MOF *Siehe* Meta Object Facility
Monkey Work 208, 240
Muster 37
  Analyse- 38, 263
  Archetyp- 38
  Architektur- 38
  Entwurfs- 38
  -orientierung 37
  -Replizierung 39
  Transformations- 28
Mustererkennung 404

O

Object Constraint Language 32, 106
  `OCLExpression` 108
Objektorientierte Analyse 263
Objektorientierte Geschäftsprozessmodellierung 124
Oblique Transformation *Siehe* Transformation, schiefe
OCL *Siehe* Object Constraint Language
Ökosystem 51

OMG
als Bottleneck der MDA 434
OMT 74
OOA *Siehe* Objektorientierte Analyse
OOD 74
OOGPM *Siehe* Objektorientierte Geschäftsprozessmodellierung
OOSE 74
Optimierung 152
Organisationseinheiten 252
`Outcome` 334, 335

P

Pageflow 333
Backing Bean 310
PageFlow
Beehive 305
Parsing 394
Partial Round-Trip Engineering 174
Pattern Matching 404
PDM *Siehe* Platform Description Model
Perspektive 24
Pilotprojekt 216
PIM *Siehe* Platform Independent Model
Platform Description Model 121, 139
Platform Independent Model 27
Platform Specific Model 27, 141
Plattform 26, 126
generisch 127
herstellergebunden 128
Stapel 26
technologiespezifisch 127
-unabhängigkeit 34
Plattform Experte 203
Plattform Independent Model 126
Pragmatismus 47, 234
Produktionsregel 169
Produktlinie 188
Profil 25
Proof-of-Concept 216
Protected Regions *Siehe* User-Sections
Prozessmodell 184
PSM *Siehe* Platform Specific Model

Q

Quellelemente
Ermittlung der 406
Quellmetamodell 162
Queries Views and Transformations 32
QVT *Siehe* Queries Views and Transformations

R

Rational 75
Refaktorisierung 151
Reflektives Programmiermodell 155
Requirements Model 123
Rumbaugh, James *Siehe* Die drei Amigos
Runaway Projects 231

S

Semantik 66
Separation of Concerns 58
Shelfware 239
Sicht 25, 99
Sketch-Modelle 63
Softeareentwicklung
adaptive 225
evolutionäre 225
Software Architekt 207
Software Entwickler 208
Software-Analyst 207
Software-Engineering 12
Softwareentwicklung
Geschichte der 11
Iterativ-Inkrementelle 187, 223
kunden-getriebene 225
Modellgetriebene 19
risiko-getriebene 225
Software-Factories 46
Basics 424
Software-Krise 10
Software-Produktlinie 133
Sofware-Factories 423

SPEM 185
Spracharchitekt 203
Sprachsymbol 169
Standish Group 11
Stereotyp 95
Subdomäne *Siehe* Domäne, Sub-
Synchronisation von Modellen
und Code 173
Syntax 66
abstrakte 68, 85
konkrete 68
Syntaxbaum 169
System 23
Fraktalität 58
Granularität 58
Querschnittseigenschaften 59
Systemfamilie 129

T

TDM *Siehe* Transformation
Description Model
Technologie Change Project 212
Technologie-Bindung *Siehe*
Technologie-Projektion
Technologie-Plan 269
Technologie-Projektion 121, 139
Template 172
Tester 209
Timeboxing 226, 228
Tool Gap 122
Transformation 27, 149
Beschreibungssprache gATL
389
Description Model 121
horizontale 152
Klassifikation von 152
mittels Jython 404
Modell-zu-Modell 150, 153,
411
Modell-zu-Text 150, 167
Muster 28
-sarchitekt 204
schiefe 153
-smetamodell 163
-sregel 155, 163
-ssprache 166
unter Einsatz universeller
Programmiersprachen 164
unter Einsatz von
Skriptsprachen 165
vertikale 152
Turing-vollständige Sprachen 66

U

Übersetzer 32
UML *Siehe* Unified Modeling
Language
UML2 (Framework) 295
Unified Modeling Language 31,
73
Action Semantics 101
-Infrastructure 80
Profile 76
-Profile 94
-Repository 98
-Spezifikation 80
-Stack 80
-Superstructure 82
Testing Profile (U2TP) 210
User-Sections 176

V

Variabilität
funktionale 133
Variabilität (von Systemfamilien)
188
Velocity 172
Verfeinerung 151
View 25
-point 24
Virtuelle Maschine 127
Visual Studio Team System 46,
423

W

Warner, Jos 114
Weltmodell 235

X

XMI *Siehe* XML Metadata
Interchange
XML Metadata Interchange 31,
81, 91
XML Metadata Interface 156

xUML *Siehe* Executable UML

Y

Yacc 395

Z

Zielaktionen
  Durchführung der 407
Zielmetamodell 162
Zustandsdiagramm 75

Literaturverzeichnis

[AB01] David H. Akehurst und Behzad Bordbar: *On Querying UML Data Models with OCL*. In: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools. Springer, 2001.

[AIS77] Christopher Alexander, Sara Ishikawa und Murray Silverstein: *A Pattern Language – Towns, Buildings, Construction*. Oxford University Press, 1977.

[AL04] John Arthorne und Chris Laffra: *Official Eclipse 3.0 FAQs*. Addison-Wesley, 2004.

[AM] Kent Beck et al.: *Manifesto for Agile Software Development*.
http://agilemanifesto.org/ (letzter Abruf Mai 2005)

[Amb03a] Scott W. Ambler: *Examining the Model Driven Architecture (MDA)*.
http://www.agilemodeling.com/essays/mda.htm (letzter Abruf Mai 2005)

[Amb03b] Scott W. Ambler: *Agile Database Techniques*. John Wiley & Sons, 2003.

[Amb04] Scott W. Ambler: *A UML Profile for Data Modeling*.
http://www.agiledata.org/essays/umlDataModelingProfile.html (letzter Abruf Mai 2005)

[AN04] Jim Arlow und Ila Neustadt: *Enterprise Patterns and MDA - Building Better Software with Archetype Patterns an UML*. Addison-Wesley, 2004.

[AndroMDA] AndroMDA.org: *Components quickly with AndroMDA*.
http://www.andromda.org/ (letzter Abruf Mai 2005)

[Ant] Apache Software Foundation (ASF): *Apache Ant – Welcome*.
http://ant.apache.org/ (letzter Abruf Mai 2005)

[ASU86] Alfred V. Aho, Ravi Sethi und Jeffrey D. Ullman: *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[ATL] ATLAS Group: *The ATL home page*.
http://www.sciences.univ-nantes.fr/lina/atl/ (letzter Abruf Mai 2005)

[Bas04] Wim Bast: *TheServerSide Debates – Software Factories vs. MDA*. (Antwort von Wim Bast an Jack Greenfield auf *TheServerSide.net*).
http://www.theserverside.net/news/thread.tss?thread_id=30082 (letzter Abruf Januar 2006)

[Bau93] Friedrich L. Bauer: *Software Engineering – wie es begann*. In: Informatik-Spektrum 16, S. 259-260, Springer, Februar 1993.

[Beehive] Apache Software Foundation (ASF): *Welcome to Apache Beehive*.
http://beehive.apache.org (letzter Abruf Januar 2006)

[Ber03] Hans Bergsten: *JavaServer Faces*. O'Reilly, 2003.

[Béz03] Jean Bézivin: *MDA™ From Hype to Hope and Reality*. In: «UML» 2003, San Francisco, Invited Talk.
http://www.sciences.univ-nantes.fr/info/perso/permanents /bezivin/UML.2003/UML.SF.JB.GT.ppt (letzter Abruf Mai 2005)

[BC87] Kent Beck und Ward Cunningham: *Using Pattern Languages for Object-Oriented Programs*. In: Technical Report CR-87-43, Tektronix, Inc., September 17, 1987.

[BHK04] Marc Born, Eckhardt Holz und Olaf Kath: *Softwareentwicklung mit UML 2 – Die neuen Entwurfstechniken UML 2, MOF 2 und MDA*. Addison-Wesley, 2004.

[BHL+04] Jean Bézivin, Slimane Hammoudi, Denivaldo Lopes und Frédéric Jouault: *An Experiment in Mapping Web Services to Implementation Platforms*.
http://www.sciences.univ-nantes.fr/info/recherche/Vie/RR/RR-IRIN2004-01.pdf (letzter Abruf Mai 2005)

[BK04] Christian Bauer und Gavin King: *Hibernate in Action – Practical Object/Relational Mapping*. Manning, 2004.

[BKPS04] Günter Böckle, Peter Knauber, Klaus Pohl und Klaus Schmid (Hrsg.): *Software-Produktlinien – Methoden, Einführung und Praxis*. dpunkt, 2004.

[BL74] Walter S. Brainerd und Lawlrence H. Landweber: *Theory of Computation*. John Wiley & Sons, 1974.

[Boc03a] Conrad Bock: *UML 2 Activity and Action Models*. In: Journal of Object Technology, Volume 2, Number 4, S. 43-53, ETH Zürich, Juli/August 2003.
http://www.jot.fm/issues/issue_2003_07/column3/ (letzter Abruf Mai 2005)

[Boc03b] Conrad Bock: *UML 2 Activity and Action Models Part 2 – Actions*. In: Journal of Object Technology, Volume 2, Number 5, S. 41-56, ETH Zürich, September/Oktober 2003.
http://www.jot.fm/issues/issue_2003_09/column4/ (letzter Abruf Mai 2005)

[Boc03c] Conrad Bock: *UML 2 Activity and Action Models Part 3 – Control Nodes*. In: Journal of Object Technology, Volume 2, Number 6, S. 7-23, ETH Zürich, November/Dezember 2003.
http://www.jot.fm/issues/issue_2003_11/column1/ (letzter Abruf Mai 2005)

[Boc04a] Conrad Bock: *UML 2 Activity and Action Models Part 4 – Object Nodes*. In: Journal of Object Technology, Volume 3, Number 1, S. 27-41, ETH Zürich, Januar/Februar 2004.
http://www.jot.fm/issues/issue_2004_01/column3/ (letzter Abruf Mai 2005)

[Boc04b] Conrad Bock: *UML 2 Activity and Action Models Part 5 – Partitions*. In: Journal of Object Technology, Volume 3, Number 7, S. 37-56, ETH Zürich, Juli/August 2004.
http://www.jot.fm/issues/issue_2004_07/column4/ (letzter Abruf Mai 2005)

[Boe76] Barry W. Boehm: *Software Engineering*. In: IEEE Transactions on Computers, Volume 25, Number 12, S. 1226-1241, IEEE Computer Society, Dezember 1976.

[Boe95] Barry W. Boehm: *Anchoring the Software Process*. USC CSE Technical Reports, 1995.
http://sunset.usc.edu/publications/TECHRPTS/1995/usccse95-507/ASP.pdf (letzter Abruf Mai 2005)

[Boo93] Grady Booch: *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 1993.

[Boo04a] Grady Booch: *Microsoft and Domain Specific Languages* (Blog-Eintrag auf *Handbook of Software Architecture*).
http://www.booch.com/architecture/blog.jsp?archive=2004-00.html (letzter Abruf Januar 2006)

[Boo04b] Grady Booch: *Expansion of the UML* (Blog-Eintrag auf *developerWorks : Blogs : Grady Booch*).
www-128.ibm.com/developerworks/blogs/dw_blog.jspa?roll=-20&blog=317 (letzter Abruf Januar 2006)

[Bos01] Jan Bosch: *Software Product Lines – Organizational Alternatives*. In: Proceedings of the 23rd International Conference on Software Engineering (ICSE'01), S. 91-100, IEEE Computer Society, 2001.

[BPEL] International Business Machines Corporation (IBM) et al.: *Business Process Execution Language for Web Services v1.1*.
http://www-128.ibm.com/developerworks/library/specification/ws-bpel/ (letzter Abruf November 2005)

[BPELEXT] JBoss Inc.: *jBPM BPEL Extension*.
http://jira.jboss.com/jira/browse/BPEL (letzter Abruf November 2005)

[BSM+03] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick und Timothy J. Grose: *Eclipse Modeling Framework*. Addison-Wesley, 2003.

[CASE] Wikipedia: *Computer-aided software engineering*.
http://en.wikipedia.org/wiki/CASE (letzter Abruf Mai 2005)

[CC] ThoughtWorks Inc.: *CruiseControl Home*.
http://cruisecontrol.sourceforge.net/ (letzter Abruf Mai 2005)

[CCM] Object Management Group (OMG): *CORBA Component Model*.
http://www.omg.org/technology/documents/formal/components.htm (letzter Abruf Mai 2005)

[CE00] Krysztof Czarnecki und Ulrich W. Eisenecker: *Generative Programming – Methods, Tools and Applications*. Addison-Wesley, 2000.

[CF] Wikipedia: *Creeping featurism*.
http://en.wikipedia.org/wiki/Creeping_featurism (letzter Abruf Mai 2005)

[CGN] Code Generation Network: *Code Generation Information for the Pragmatic Engineer*.
http://www.codegeneration.net/ (letzter Abruf Mai 2005)

[CI] Martin Fowler und Matthew Foemmel: *Continuous Integration*.
http://www.martinfowler.com/articles/continuousIntegration.html (letzter Abruf Mai 2005)

[CK92] Michael G. Christel und Kyo C. Kang: *Issues in Requirements Elicitation*. Carnegie Mellon University/Software Engineering Institute, 1992.
http://www.sei.cmu.edu/publications/documents/92.reports/92.tr.012.html (letzter Abruf Januar 2006)

[CN01] Paul Clemens und Linda Northrop: *Software Product Lines – Practices and Patterns*. Addison-Wesley, 2001.

[Coc01] Alistair Cockburn: *Agile Software Development*. Addison-Wesley, 2001.

[Coc04] Alistair Cockburn: *Crystal Clear – A Human-Powered Methodology for Small Teams*. Addison-Wesley, 2004.

[Col95] Andy Cole: *Runaway Projects – Cause and Effects*. In: Software World, Volume 26, Number 3, S. 3-5, A. P. Publications, 1995.

[CookBlog] Steve Cook: *Steve Cook´s WebLog*.
http://blogs.msdn.com/stevecook/default.aspx (letzter Abruf Januar 2006)

[Copyleft] Wikipedia: *Copyleft*.
http://en.wikipedia.org/wiki/Copyleft (letzter Abruf Mai 2005)

[COTS] Wikipedia: *Commercial off-the-shelf*.
http://en.wikipedia.org/wiki/COTS (letzter Abruf Mai 2005)

[COWO00] COMPUTERWOCHE.de: *Java-Projekt Cheops in den Sand gesetzt*.
http://www.computerwoche.de/index.cfm?artid=17266 (letzter Abruf Mai 2005)

[CPL10] Eclipse.org: *Common Public License - v 1.0*. http://www.eclipse.org/legal/cpl-v10.html (letzter Abruf Mai 2005)

[CPLFAQ] International Business Machines Corporation (IBM): *Common Public License (CPL) Frequently asked questions*. http://www-106.ibm.com/developerworks/library/os-cplfaq.html (letzter Abruf Mai 2005)

[CRUD] Wikipedia: *CRUD*. http://en.wikipedia.org/wiki/CRUD_(acronym) (letzter Abruf März 2006)

[CT03] Detlef Borchers: *Verursacherbedingt verspätet – Das „fortschrittlichste Mautsystem der Welt" und die Realität*. In: c´t 22/2003, Seite 92, Heise Zeitschriften Verlag, 2003.

[CWH00] Mary Campione, Kathy Walrath und Alison Huml: *The Java Tutorial, Third Edition*. Addison-Wesley, 2000. http://java.sun.com/docs/books/tutorial/ (letzter Abruf Mai 2005)

[DBC] Wikipedia: *Design by Contract*. http://en.wikipedia.org/wiki/Design_by_contract (letzter Abruf Juli 2005)

[DDH72] Ole-Johan Dahl, Edsger W. Dijkstra und C. A. R. Hoare: *Structed Programming*. Academic Press, 1972.

[DDL] Wikipedia: *Data Definition Language*. http://en.wikipedia.org/wiki/Data_Definition_Language (letzter Abruf Mai 2005)

[DE] The Software Engineering Institute (SEI): *Domain Engineering: A Model-Based Approach*. http://www.sei.cmu.edu/domain-engineering/ (letzter Abruf Mai 2005)

[DeM78] Tom DeMarco: *Structured Analysis and System Specification*. Yourdon Press, 1978.

[DeM01] Tom DeMarco: *Slack – Getting Past Burnout, Busywork, and the Myth of Total Efficiency*. Broadway Books, 2001.

[DL99] Tom DeMarco und Timothy Lister: *Peopleware – Productive Projects and Teams, Second Edition*. Dorset House, 1999.

[DL03] Tom DeMarco und Timothy Lister: *Waltzing with Bears – Managing Risk on Software Projects*. Dorset House, 2003.

[DSL] Wikipedia: *Domain-Specific Language*. http://en.wikipedia.org/wiki/Domain_Specific_Language (letzter Abruf Mai 2005)

[EBNF] Wikipedia: *Extended Backus-Naur form*. http://en.wikipedia.org/wiki/Extended_Backus-Naur_form (letzter Abruf Mai 2005)

[Eclipse] Eclipse.org: *Eclipse.org Main Page*. http://www.eclipse.org/ (letzter Abruf Mai 2005)

[EJB] Sun Microsystems: *Enterprise JavaBeans Technology*. http://java.sun.com/products/ejb/ (letzter Abruf Mai 2005)

[EMF] Eclipse.org: *EMF – Eclipse Modeling Framework*. http://www.eclipse.org/emf (letzter Abruf Mai 2005)

[EMFFAQ] Eclipse.org: *Eclipse Modeling Framework FAQ*. http://www.eclipse.org/emf/faq (letzter Abruf Mai 2005)

[EMFvsMDR] Martin Matula: *EMF vs MDR*. http://mdr.netbeans.org/servlets/ReadMsg?list=users&msgNo=1877 (letzter Abruf Mai 2005)

[ENEWS] Eclipse.org: *Eclipse newsgroups home*. http://www.eclipse.org/newsgroups/ (letzter Abruf November 2005)

[ETP] Eclipse.org: *Eclipse Tools Project*. http://www.eclipse.org/tools (letzter Abruf Mai 2005)

[EUML2a] Eclipse.org: *The Eclipse UML2 Project*. http://www.eclipse.org/uml2 (letzter Abruf Mai 2005)

[EUML2b] Kenn Hussey: *Getting Started with UML2*. http://dev.eclipse.org/viewcvs/indextools.cgi/~checkout~/uml2-home/docs/articles/Getting_Started_with_UML2/article.html (letzter Abruf November 2005)

[EUML2c] Kenn Hussey: *Introduction to UML2 Profiles* http://dev.eclipse.org/viewcvs/indextools.cgi/~checkout~/uml2-home/docs/articles/Introduction_to_UML2_Profiles/article.html (letzter Abruf November 2005)

[EUML2d] Eclipse.org: *Overview (Unified Modeling Language 2.0 Javadoc)*. http://download.eclipse.org/tools/uml2/1.1.1/javadoc/ (letzter Abruf November 2005)

[Eva03] Eric Evans: *Domain-Driven Design – Tackling Complexity in the Heart of Software*. Addison-Wesley, 2003.

[Fei06] Rich Feit: *Integrating JavaServer Faces with Beehive PageFlow*. http://dev2dev.bea.com/pub/a/2005/12/integrating-jsf-beehive.html (letzter Abruf Januar 2006)

[Fow96] Martin Fowler: *Analysis Patterns – Reusable Object Models*. Addison-Wesley, 1996.

[Fow99] Martin Fowler: *Refactoring – Improving The Design Of Existing Code*. Addison-Wesley, 1999.

[Fow04a] Martin Fowler: *UmlAsSketch*. http://www.martinfowler.com/bliki/UmlAsSketch.html (letzter Abruf Mai 2005)

[Fow04b] Martin Fowler: *UnwantedModelingLanguage*. http://www.martinfowler.com/bliki/UnwantedModelingLanguage.html (letzter Abruf Mai 2005)

[Fra03] David S. Frankel: *Model Driven Architecture – Applying MDA To Enterprise Computing*. John Wiley & Sons, 2003.

[Fra04] David S. Frankel: *Software Industrialization and the New IT*. In: David S. Frankel und John Parodi (Hrsg.): The Mda Journal – Model Driven Architecture Straight From the Masters. Meghan-Kiffer, 2004.

[Fra05] David S. Frankel: *Eclipse and MDA – The Expanding Significance of the Eclipse Modeling Framework (EMF)*. In: MDA Journal – A BPT Column, Business Process Trends (www.bptrends.com), März 2005. http://www.bptrends.com/publicationfiles/03-05 COL Eclipse and MDA - Frankel1.pdf (letzter Abruf Mai 2005)

[Gallup05] The Gallup Organization: *Engagement Index 2005 – Engagement der ArbeitnehmerInnen in Deutschland verharrt auf niedrigem Niveau*. http://www.presseportal.de/story.htx?nr=719311 (letzter Abruf November 2005)

[GB03] Erich Gamma und Kent Beck: *Contributing to Eclipse – Principles, Patterns and Plug-Ins*. Addison-Wesley, 2003.

[GBM03] David Gallardo, Ed Burnette und Robert McGovern: *Eclipse in Action – A Guide for Java Developers*. Manning, 2003.

[GC03] Joseph D. Gradecki und Jim Cole: *Mastering Apache Velocity*. John Wiley & Sons, 2003.

[GDB03] Timothy J. Grose, Gary C. Doney und Steven Brodsky: *Mastering XMI – Java Programming with the XMI, XML, and UML*. John Wiley & Sons, 2003.

[GEF] Eclipse.org: *Graphical Editing Framework.* http://www.eclipse.org/gef (letzter Abruf Mai 2005)

[GHH04] David Geary, Cay Horstmann und Cay S. Horstmann: *Core JavaServer Faces.* Prentice Hall, 2004.

[Gla99] Robert L. Glass: *The Realities of Software Technology Payoffs*. In: Communications of the ACM, Volume 42, Issue 2, S. 74-79, ACM, 1999.

[Gla02] Robert L. Glass: *Facts and Fallacies of Software Engineering*. Addison-Wesley, 2002.

[GOF] Erich Gamma, Richard Helm, Ralph E. Johnson und John Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gom04] Hassan Gomaa: *Designing Software Product Lines with UML – From Use Cases to Pattern-Based Software Architectures.* Addison-Wesley, 2004.

[GP] Krysztof Czarnecki und Ulrich W. Eisenecker: *Generative Programming.* http://www.generative-programming.org/ (letzter Abruf Mai 2005)

[GP04] Gunnar Overgaard und Karin Palmkvist: *Use Cases : Patterns and Blueprints.* Addison-Wesley, 2004.

[Gra00] Graeme Smith: *The Object-Z Specification Language – Advances in Formal Methods*. Kluwer, 2000.

[GreenBlog] Jack Greenfield: *Jack Greenfield´s Blog*. http://blogs.msdn.com/jackgr/ (letzter Abruf Januar 2006)

[Groovy] The Codehaus: *Groovy – Home*. http://groovy.codehaus.org/ (letzter Abruf Mai 2005)

[GSCK04] Jack Greenfield, Keith Short, Steve Cook und Stuart Kent: *Software Factories – Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, 2004.

[Har87] David Harel: *Statecharts – A visual Formalism for Complex Systems*. In: Science of Computer Programming, Volume 8, Issue 3, S. 231-274, Elsevier, Juni 1987.

[Hay96] David C. Hay: *Data Model Patterns – Conventions of Thought*. Dorset House, 1996.

[Hibernate] JBoss Inc.: *Hibernate – Relational Persistence For Idiomatic Java*.
http://www.hibernate.org/ (letzter Abruf November 2005)

[HM04] Sven Haiges (Hrsg.) und Marcel May: *JavaServer Faces - Web Development mit dem Standard-Framework*. Software & Support, 2004.

[HO04] heise online: *Bericht – Software-Projekt für Finanzämter gescheitert*.
http://www.heise.de/newsticker/meldung/48843 (letzter Abruf Mai 2005)

[HP] Portland Pattern Repository's Wiki: *Hollywood Principle*.
http://c2.com/cgi/wiki?HollywoodPrinciple (letzter Abruf Mai 2005)

[HS99] Peter Herzum und Oliver Sims: *Business Component Factory – A Comprehensive Overview of Component-Based Development for the Enterprise*. John Wiley & Sons, 1999.

[Hub01] Richard Hubert: *Convergent Architecture – Building Model-Driven J2EE Systems with UML*. John Wiley & Sons, 2001.

[HUTN] Object Management Group (OMG): *Human-Usable Textual Notation (HUTN) v1.0*.
http://www.omg.org/docs/formal/04-08-01.pdf (letzter Abruf Mai 2005)

[IEEE1471] IEEE Computer Society: *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems [IEEE Std 1471-2000]*. IEEE Computer Society, 2000.

[Jakarta] Apache Software Foundation (ASF): *The Apache Jakarta Project*.
http://jakarta.apache.org/ (letzter Abruf Mai 2005)

[JavaCC] java.net: *JavaCC – Documentation Index*.
http://javacc.dev.java.net/doc/docindex.html (letzter Abruf Mai 2005)

[JavaEE] Sun Microsystems: *Java Platform, Enterprise Edition (Java EE)*.
http://java.sun.com/j2ee/ (letzter Abruf Mai 2005)

[JBCR01] Jim Johnson, Karen D. Boucher, Kyle Connors und James Robinson: *Collaborating on Project Success*. In: Software Magazine, Wiesner Publishing, Februar/März 2001.

[JBGD-GS] JBoss Inc.: *JBoss jBPM Graphical Process Designer – Getting started*.
http://docs.jboss.com/jbpm/v3/gpd/index.html (letzter Abruf November 2005)

[JBoss] JBoss Inc.: *JBoss.com – The Professional Open Source Company*.
http://www.jboss.org/index.html (letzter Abruf Mai 2005)

[JBPM] JBoss Inc.: *JBoss jBPM*.
http://www.jboss.com/products/jbpm (letzter Abruf November 2005)

[JBR98] Ivar Jacobson, Grady Booch und James Rumbaugh: *The Unified Software Development Process*. Addison-Wesley, 1998.

[JCJO92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson und Gunnar Overgaard: *Object-Oriented Software Engineering – A Use Case Driven Approach*. Addison-Wesley, 1992.

[JFC] Sun Microsystems: *Java Foundation Classes (JFC/Swing)*.
http://java.sun.com/products/jfc/ (letzter Abruf Mai 2005)

[JLS] James Gosling, Bill Joy, Guy Steele und Gilad Bracha: *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.
http://java.sun.com/docs/books/jls/download/langspec-2.0.pdf (letzter Abruf Mai 2005)

[JMI] Sun Microsystems: *Java Metadata Interface (JMI)*.
http://java.sun.com/products/jmi/ (letzter Abruf Mai 2005)

[JRH+03] Mario Jeckle, Chris Rupp, Jürgen Hahn, Barbara Zengler und Stefan Queins: *UML 2 glasklar*. Hanser, 2003.

[JSR12] The Java Community Process: *JSRs – Java Specification Requests – detail JSR# 12*.
http://www.jcp.org/en/jsr/detail?id=12 (letzter Abruf Mai 2005)

[JSR175] The Java Community Process: *JSRs – Java Specification Requests – detail JSR# 175*.

[Jython] jython.org: *Jython Home Page*.
http://www.jython.org/ (letzter Abruf Mai 2005)

[JythonAPI] jython.org: *Overview (Jython API documentation)*.
http://www.jython.org/docs/javadoc/ (letzter Abruf Mai 2005)

[JythonReg] jython.org: *The Jython Registry*.
http://www.jython.org/docs/registry.html (letzter Abruf Mai 2005)

[KentBlog] Stuart Kent: *stuart kent´s blog*.
http://blogs.msdn.com/stuart_kent/ (letzter Abruf Januar 2006)

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina, Videira Lopes, Jean-Marc Loingtier und John Irwin: *Aspect-Oriented Programming*. In: European Conference on Object-Oriented Programming, Springer, 1997.

[KWB03] Anneke Kleppe, Jos Warmer und Wim Bast: *MDA Explained. The Model Driven Architecture – Practice and Promise*. Addison-Wesley, 2003.

[Lad03] Ramnivas Laddad: *AspectJ in Action – Practical Aspect-Oriented Programming*. Manning, 2003.

[Lar04] Craig Larman: *Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd Edition*. Prentice Hall, 2004.

[LM97] Wing Lam und John A. McDermid: *A summary of domain analysis experience by way of heuristics*. In: Proceedings of the 1997 symposium on Software reusability, S. 54-64, ACM, 1997.

[M2T-RFP] Object Management Group (OMG): *MOF Model to Text Transformation Language – Request For Proposal*.
http://www.omg.org/docs/ad/04-04-07.pdf

[MB02] Stephen J. Mellor und Marc J. Balcer: *Executable UML – A Foundation for Model Driven Architecture*. Addison-Wesley, 2002.

[MBK04] Craig McClanahan, Ed Burns und Roger Kitain: *JavaServer Faces Specification – Version 1.1*. Sun Microsystems, 2004.
http://java.sun.com/j2ee/javaserverfaces/download.html (letzter Abruf Mai 2005)

[MDAATP] Object Management Group (OMG): *Model Driven Architecture – A Technical Perspective*.
http://www.omg.org/cgi-bin/apps/doc?ormsc/01-07-01.pdf (letzter Abruf Mai 2005)

[MDAGuide] Object Management Group (OMG): *MDA Guide Version 1.0.1*.
http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf (letzter Abruf Mai 2005)

[MDG+04] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht und Philippe Vanderheyden: *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. International Business Machines Corporation, 2004.
http://www.redbooks.ibm.com/redbooks/pdfs/sg246302.pdf (letzter Abruf Mai 2005)

[MDR] NetBeans.org: *Metadata Repository home*.
http://mdr.netbeans.org/ (letzter Abruf Mai 2005)

[MOF2] Object Management Group (OMG): *Meta Object Facility (MOF) 2.0 Core Final Adopted Specification*.
http://www.omg.org/docs/ptc/03-10-04.pdf (letzter Abruf Mai 2005)

[MOF2IDL] Object Management Group (OMG): *MOF 2.0 IDL Final Adopted Specification*.
http://www.omg.org/docs/ptc/04-07-01.pdf (letzter Abruf Mai 2005)

[MOF2XMI] Object Management Group (OMG): *MOF 2.0/XMI Mapping Specification, v2.1*.
http://www.omg.org/docs/formal/05-09-01.pdf (letzter Abruf September 2005)

[MW] Wikipedia: *Middleware*.
http://de.wikipedia.org/wiki/Middleware (letzter Abruf Mai 2005)

[NetBeans] NetBeans.org: *Welcome to NetBeans*.
http://www.netbeans.org/ (letzter Abruf Mai 2005)

[Nor04a] Theodore S. Norvell: *The JavaCC Tutorial*. http://www.engr.mun.ca/~theo/JavaCC-Tutorial/ (letzter Abruf Mai 2005)

[Nor04b] Theodore S. Norvell: *The JavaCC FAQ*. http://www.engr.mun.ca/~theo/JavaCC-FAQ/ (letzer Abruf Mai 2005)

[NRB76] Peter Naur, Brian Randell und John N. Buxton (Hrsg.): *Software Engineering – Concepts and Techniques – Proceedings of the NATO Conferences*. Petrocelli/Charter, 1976.

[OCL2] Object Management Group (OMG): *UML 2.0 OCL Final Adopted Specification*. http://www.omg.org/docs/ptc/03-10-14.pdf (letzter Abruf Mai 2005)

[Oes04] Bernd Oestereich: *Analyse und Design mit UML 2 – Objektorientierte Softwareentwicklung, Siebte Auflage*. Oldenbourg, 2004.

[Oes05] Bernd Oestereich: *Objektorientierte Geschäftsprozessmodelllierung und modellgetriebene Softwareentwicklung*. In: HMD – Praxis der Wirtschaftsinformatik, Heft 241, S. 27-34, dpunkt, Februar 2005. http://www.oose.de/downloads/HMD_241_GPM_und_MDSE_Oestereich.pdf (letzter Abruf Mai 2005)

[OOGPM] oose.de: *Objektorientierte Geschäftsprozessmodellierung mit der UML – Die Homepage zum Buch*. http://www.oogpm.de (letzter Abruf November 2005)

[ORe04] Graham O'Regan: *Introduction to Aspect-Oriented Programming*. http://www.onjava.com/pub/a/onjava/2004/01/14/aop.html (letzter Abruf Mai 2005)

[OSI] Open Source Initiative (OSI): *Welcome*. http://www.opensource.org (letzter Abruf Mai 2005)

[OWS+03] Bernd Oesterreich, Christian Weiss, Claudia Schröder, Tim Weilkiens und Alexander Lenhard: *Objektorientierte Geschäftsprozessmodellierung mit der UML*. dpunkt, 2003.

[Par72] David L. Parnas: *On the Criteria to Be Used in Decomposing Systems into Modules*. In: Communications of the ACM, Volume 15, Issue 12, S. 1053-1058, ACM, Dezember 1972.

[Pen03] Tom Pender: *UML Bible*. John Wiley & Sons, 2003.

[PG04] Eric Pugh und Joseph D. Gradecki: *Professional Hibernate*. John Wiley & Sons, 2004.

[POC] Wikipedia: *Proof of concept*.
http://en.wikipedia.org/wiki/Proof_of_concept (letzter Abruf Mai 2005)

[POJO] Portland Pattern Repository's Wiki: *Plain Old Java Object*.
http://c2.com/cgi/wiki?PlainOldJavaObject (letzter Abruf Mai 2005)

[Pop04a] Remko Popma: *JET Tutorial Part 1 (Introduction to JET)*.
http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html (letzter Abruf Mai 2005)

[Pop04b] Remko Popma: *JET Tutorial Part 2 (Write Code that Writes Code)*.
http://www.eclipse.org/articles/Article-JET2/jet_tutorial2.html (letzter Abruf Mai 2005)

[POSA1] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad und Michael Stal: *Pattern-Oriented Software Architecture – Volume 1 – A System of Patterns*. John Wiley & Sons, 1996.

[PR06] Daniel Pieper und Carsten Röttgers: *MDA mit der UML 2.0 – Neue Möglichkeiten der Verhaltensmodellierung.* In: OBJEKTspektrum 01/2006, S. 46-51. SIGS-DATACOM, Januar 2006.

[Pragmatism] Wikipedia: *Pragmatism*.
http://en.wikipedia.org/wiki/Pragmatism (letzter Abruf Mai 2005)

[Python] python.org: *Python Programming Language*.
http://www.python.org/ (letzter Abruf Mai 2005)

[Qui94] Klaus Quibeldey-Cirkel: *Paradigmenwechsel im Software-Engineering – Auf dem Weg zu objektorientierten Weltmodellen.* In: Softwaretechnik Trends 14/1, S. 47–58, Gesellschaft für Informatik, 1994.

[QVT] Object Management Group (OMG): *MOF QVT Final Adopted Specification*.
http://www.omg.org/docs/ptc/05-11-01.pdf (letzter Abruf November 2005)

[QVT-RFP] Object Management Group (OMG): *MOF 2.0 Query / Views / Transformations RFP*. http://www.omg.org/docs/ad/02-04-10.pdf (letzter Abruf Mai 2005)

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy und William Lorensen: *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[Ric06] Chris Richardson: *POJOs in Action – Developing Enterprise Applications with Lightweight Frameworks*. Manning, 2006.

[RJB04] James Rumbaugh, Ivar Jacobson und Grady Booch: *The Unified Modeling Language Reference Manual, Second Edition*. Addison-Wesley, 2004.

[ROSE] International Business Machines Corporation (IBM): *Rational Rose XDE Developer – Family Overview*. http://www-306.ibm.com/software/awdtools/developer/rosexde (letzter Abruf Mai 2005)

[SDL] ITU-T: *Specification and Description Language (SDL) [ITU-T-Z.100]*. ITU-T, 2002.

[SF03] Howard Smith und Peter Fingar: *IT Doesn't Matter – Business Processes Do – A Critical Analysis of Nicholas Carr's I.T. Article in the Harvard Business Review*. Meghan-Kiffer, 2003.

[SG03] The Standish Group: *Latest Standish Group CHAOS Report Shows Project Success Rates Have Improved by 50%*. http://www.standishgroup.com/press/article.php?id=2 (letzter Abruf Mai 2005)

[SGW94] Bran Selic, Garth Gullekson und Paul T. Ward: *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.

[SHNM04] Matthew Scarpino, Stephen Holder, Stanford Ng und Laurent Mihalkovic: *SWT/JFace in Action – GUI Design with Eclipse 3.0*. Manning, 2004.

[ShortBlog] Keith Short: *Keith Short's WebLog*. http://blogs.msdn.com/keith_short/ (letzter Abruf Januar 2006)

[SPEM] Object Management Group (OMG): *Sofware Process Engineering Metamodel, Version 1.1*. http://www.omg.org/technology/documents/formal/spem.htm (letzter Abruf Mai 2005)

[SPL03] Robert C. Seacord, Daniel Plakosh und Grace A. Lewis: *Modernizing Legacy Systems – Software Technologies, Engineering Processes, and Business Practices*. Addison Wesley, 2003.

[Tay95] David A. Taylor: *Business Engineering with Object Technology*. John Wiley & Sons, 1995.

[TMC03] The Middleware Company (TMC): *Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) Approach – Productivity Analysis.* http://www.omg.org/mda/mda_files/MDA_Comparison-TMC_final.pdf (letzter Abruf November 2005)

[TPTP] Eclipse.org: *Eclipse Test & Performance Tools Platform.* http://www.eclipse.org/tptp/index.html (letzter Abruf September 2005)

[U2TP] The U2TP Consortium: *U2TP – The UML Testing Profile*. http://www.fokus.gmd.de/u2tp/ (letzter Abruf September 2005)

[UML] Object Management Group (OMG): *UML Resource Page.* http://www.omg.org/uml/ (letzter Abruf Mai 2005)

[UML2IS] Object Management Group (OMG): *UML 2.0 Infrastructure Final Adopted Specification.* http://www.omg.org/docs/ptc/04-10-14.pdf (letzter Abruf November 2005)

[UML2SS] Object Management Group (OMG): *UML 2.0 Superstructure Final Adopted Specification.* http://www.omg.org/docs/ptc/05-07-04.pdf (letzter Abruf November 2005)

[UMLProfiles] Object Management Group (OMG): *MDA Specifications – UML Profiles.* http://www.omg.org/mda/specs.htm#Profiles (letzter Abruf Mai 2005)

[UPfEDOC] Object Management Group (OMG): *UML Profile For Enterprise Distributed Object Computing (EDOC).* http://www.omg.org/technology/documents/formal/edoc.htm (letzter Abruf Mai 2005)

[UPfEJB] Java Community Process: *JSR 26 – UML/EJB Mapping Specification*.
http://www.jcp.org/aboutJava/communityprocess/review/jsr026/ (letzter Abruf Mai 2005)

[Velocity] Apache Software Foundation (ASF): *Velocity*.
http://jakarta.apache.org/velocity/ (letzter Abruf Mai 2005)

[Ver02] Gerhard Versteegen (Hrsg.): *Software-Management*. Springer, 2002.

[VM] Wikipedia: *Virtual Machine*.
http://en.wikipedia.org/wiki/Virtual_Machine (letzter Abruf Mai 2005)

[VSTS] Microsoft Corporation: *Microsoft Visual Studio Team System*.
http://msdn.microsoft.com/vstudio/teamsystem/default.aspx (letzter Abruf Januar 2006)

[VSTSFAQ] Microsoft Corporation: *Visual Studio 2005 Team System Modeling Strategy and FAQ*.
http://msdn.microsoft.com/vstudio/default.aspx?pull=/library/en-us/dnvs05/html/vstsmodel.asp (letzter Abruf Januar 2006)

[WeLa99] David M. Weiss und Chi Tau Robert Lai: *Software Product-Line Engineering – A Family-Based Software Development Process*. Addison-Wesley, 1999.

[Wil04] Alan Cameron Wills: *Why not base domain specific languages on UML?*
http://blogs.msdn.com/alan_cameron_wills/archive/2004/11/11/255831.aspx (letzter Abruf Januar 2006)

[WK03] Jos Warmer und Anneke Kleppe: *The Object Constraint Language – Getting Your Models Ready for MDA, Second Edition*. Addison-Wesley, 2003.

[XML] World Wide Web Consortium (W3C): *Extensible Markup Language (XML)*.
http://www.w3.org/XML/ (letzter Abruf Mai 2005)

[XS] World Wide Web Consortium (W3C): *XML Schema*.
http://www.w3.org/XML/Schema (letzter Abruf Mai 2005)

[Yacc] compilertools.net: *The Lexx and Yacc page – Yacc*.
http://dinosaur.compilertools.net/#yacc (letzter Abruf Mai 2005)

[YC79] Edward Yourdon und Larry L. Constantine: *Structured Design – Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice Hall, 1979.

[You02] Edward Yourdon: *Managing High-Intensity Internet Projects*. Prentice Hall, 2002.

[You04] Edward Yourdon: *Outsource – Competing in the Global Productivity Race*. Prentice Hall, 2004.